Probabilistic Fuzzy Logic Framework in Reinforcement Learning for Decision Making

WILLIAM HINOJOSA

School of Computing, Science and Engineering
University of Salford, Salford, UK

Submitted in Partial Fulfilment of the Requirements of the Degree of Doctor of Philosophy, September 2010

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation is the product of many years of enjoyable effort and is the result of not only the hard work of one individual, but of many, each one being equally important for its correct development.

To start with, I would like to thank my supervisor, Dr. Samia Nefti, for her incredible productive collaboration, critics, guidance, and patience. I thank her for generously sharing her talents, energy, and good advice.

I was especially lucky to have the collaboration Dr. Uzay K aymak from the Econometric Institute, Erasmus School of Economics, Erasmus University at Rotterdam, whose wise comments and advice greatly contributed to the development of the theoretical part of this work.

I thank my family; my parents, who never stopped supporting me, I thank their patience, commitment, support, and advice when I needed the most. Also, to my beloved sister Johanna, for her constant support and affection. They always believed in me and pushed me forward. Certainly, their love and support have made me the person I am today.

Big thanks go to my beautiful partner Lucy, whose love and affection made all my time here much more enjoyable. I thank her for teaching me what I needed to know about this wonderful country, for supporting and understanding me and for tolerating my irregular schedules and my frequent lateness. I am also thankful for her help in proof reading this dissertation.

A special acknowledge goes to one of my closest and best friends, Alvaro Vallejos, who greatly helped me in the completion of this dissertation. His expert advice greatly helped me to quickly and effectively solve several MS Office crises in different moments. I wouldn't have done it without his help... at least not with such a nice format!

I will not forget that I've greatly enjoyed these last years because all my wonderful friends and colleagues, who enriched my years at Salford University: Xavier Sole, Ioannis Sarakoglou, Elmabruk Abulgasem, Ahmed Al-Dulaimy, May Bunne, Nelson Costa, Antonio Espingardeiro, Indra Riady and Rosidah Sam. Thanks for always being there, for the good times, for the moments of laughter, for all the ideas we share and all the discussions we had. Their presence and constant support transformed the studies that this project concludes into an unforgettable experience. They all became a family in this journey.

Finally, I am grateful to all the members of the School of Computer Science and Engineering who kindly helped me in many different ways, especially to Lynn Crankshaw and Ruth Breckill. Finally I thank The University of Salford for providing a fantastic environment for doing my research.

ACRONYMS

AI	Artificial Intelligence
DP	Dynamic Programming
FIS	Fuzzy Inference System
IR	Infra-red
MC	Monte Carlo Methods
MDP	Markov Decision Process
NN	Neural Networks
PFL	Probabilistic Fuzzy Logic
POMDP	Partially Observable Markov Decision Process
RL	Reinforcement Learning
SARSA	State-Action-Reward-State-Action
TD	Temporal Difference

GLOSSARY

Actor-critic	Refers to a class of agent architectures, where the actor plays out a particular policy, while the critic learns to evaluate actor's policy. Both the actor and critic are simultaneously improving by bootstrapping on each other.
Agent	A system that is embedded in an environment. The controller or decision-making entity choosing actions and learning to perform a task. Examples include mobile robots, software agents, or industrial controllers.
Average-reward methods	A framework where the agent's goal is to maximize the expected payoff per step. Average-reward methods are appropriate in problems where the goal is to maximize the long-term performance. They are usually much more difficult to analyze than discounted algorithms.
Discount factor	A scalar value between 0 and 1 which determines the present value of future rewards. If the discount factor is 0, the agent is concerned with maximizing immediate rewards. As the discount factor approaches 1, the agent takes more future rewards into account. Algorithms which discount future rewards include Q-learning and TD(lambda)

Discounting	If rewards received in the far future are worth less than rewards received sooner, they are des cribed as being discounted. Humans and animals appear to discount future rewards hyperbolically; exponential discounting is common in engineering and finance.
Dynamic programming	A collection of calculation techniques for finding a policy that maximises reward or minimises costs. Is a class of solution methods for solving sequential decision problems with a compositional cost structure.
Environment	The external system that an agent is "embedded" in, and can perceive and act on.
Episode	A time segment of learning with task dependent starting and ending conditions.
Function approximation	Refers to the problem of inducing a function from training examples. Standard approximators include decision trees, neural networks, and nearest-neighbour methods.
Markov chain	A model for a random process that evolves over time such that the states (like locations in a maze) occupied in the future are independent of the states in the past given the current state.
Markov decision problem	A model for a controlled random process in which an agent's choice of action determines the probabilities of transitions of a Markov chain and lead to rewards (or

	costs) that need to be maximised (or minimised). Essentially, the outcome of applying an action to a state depends only on the current action and state (and not on preceding actions or states)
Model	The agent's view of the environment, which maps state-action pairs to probability distributions over states. Note that not every reinforcement learning agent uses a model of its environment. Basically it's a mathematical description of the environment.
Model-based algorithms	These compute value functions using a model of the system dynamics. Adaptive Real-time DP (ARTDP) is a well-known example of a model-based algorithm.
Model-free algorithms	These directly learn a value function without requiring knowledge of the consequences of doing actions. Q-learning is the best known example of a model-free algorithm.
Monte Carlo methods	A class of methods for learning value functions, which estimates the value of a state by running many trials starting at that state, then averages the total rewards received on those trials.
Policy	The decision-making function of the agent, which represents a mapping from situations to actions. Can be considered a deterministic or stochastic scheme for choosing an action at every state or location.
Policy evaluation	Determining the value of each state for a given policy.

Policy improvement	Forming a new policy that is better than the current one.
Policy iteration	Alternating steps of policy evaluation and policy improvement to converge to an optimal policy.
POMDP	Partially observable Markov decision problem. State information is available only through a set of observations.
Return	The cumulative (discounted) reward for an entire episode.
Reward	An immediate, possibly stochastic, payoff that results from performing an action in a state represented by a numerical signal to the learning agent indicating task progress or completion or the degree to which a state or action is desirable. Reward functions can be used to specify a wide range of planning goals
Sensor	Agents perceive the state of their environment using sensors, which can refer to physical transducers, such as ultrasound, or simulated feature-detectors.
State	This can be viewed as a summary of the past history of the system, which determines its future evolution.
Temporal difference algorithms	A class of learning methods, based on the idea of comparing temporally successive predictions. Possibly the single most fundamental idea in all of reinforcement learning.

Temporal difference prediction error	A measure of the inconsistency between estimates of the value function at two successive states. This prediction error can be used to improve the predictions and also to choose good actions.
Unsupervised learning	The area of machine learning in which an agent learns from interaction with its environment, rather than from a knowledgeable teacher that specifies the action the agent should take in any given state.
Value function	Is a mapping from states to real numbers, where the value of a state represents the long-term reward achieved starting from that state, and executing a particular policy. The key distinguishing feature of RL methods is that they learn policies indirectly, by instead learning value functions. RL methods can be contrasted with direct optimization methods, such as genetic algorithms (GA), which attempt to search the policy space directly. A function defined over states, which gives an estimate of the total (possibly discounted) reward expected in the future, starting from each state, and following a particular policy.
Value iteration	A single iteration of policy evaluation followed by policy improvement.

"Whether or not you think you can do something, you are probably right"

–Henry Ford.

To my parents.
For their help in pursuing my dreams for
so long, so far away from home...

ABSTRACT

This dissertation focuses on the problem of uncertainty handling during learning by agents dealing in stochastic environments by means of reinforcement learning. Most previous investigations in reinforcement learning have proposed algorithms to deal with the learning performance issues but neglecting the uncertainty present in stochastic environments.

Reinforcement learning is a valuable learning method when a system requires a selection of actions whose consequences emerge over long periods for which input-output data are not available. In most combinations of fuzzy systems with reinforcement learning, the environment is considered deterministic. However, for many cases, the consequence of an action may be uncertain or stochastic in nature. This work proposes a novel reinforcement learning approach combined with the universal function approximation capability of fuzzy systems within a probabilistic fuzzy logic theory framework, where the information from the environment is not interpreted in a deterministic way as in classic approaches but rather, in a statistical way that considers a probability distribution of long term consequences.

The generalized probabilistic fuzzy reinforcement learning (GPFRL) method, presented in this dissertation, is a modified version of the actor-critic learning architecture where the learning is enhanced by the introduction of a probability measure into the learning structure where an incremental gradient descent weight-updating algorithm provides convergence.

Experiments were performed on simulated and real environments based on a travel planning spoken dialogue system. Experimental results provided evidence to support the following claims: first, the GPFRL have shown a robust performance when used in control optimization tasks. Second, its learning speed outperforms most of other similar methods. Third, GPFRL agents are feasible and promising for the design of adaptive behaviour robotics systems.

Chapter 1

Introduction

1.1 Motivation

Learning algorithms can tackle problems where pre-programmed solutions are difficult or impossible to design. Depending on the level of available information, learning agents can apply one or more types of learning. According to the connectionist learning approach (Hinton, 1989), these algorithms are mainly of three kinds: supervised, unsupervised and reinforced learning. Unsupervised learning is suitable when target information is not available and the agent tries to form a model based on clustering or association amongst data. Supervised learning is much more powerful, but it requires the knowledge of output patterns corresponding to input data. However, in dynamic environments, where the outcome of an action is not immediately known and is subject to change, correct target data may not be available at the moment of learning, which implies that supervised approaches cannot be applied. In these environments, reward information, whose availability can be only sparse, may be the best signal that an agent receives. For such systems, reinforcement learning (RL) has proven to be a

more a ppropriate meth od than supe rvised or u nsupervised meth ods w hen the systems require a selection of actions whose consequences emerge over long periods for which input-output data are not available (Berenji and Khedkar, 1992).

A RL problem can be defined as a decision process where the agent learns how to select an action based on feedback from the environment. It can be said that the agent learns a policy that maps states of the environment into actions. Often, the RL agent must learn a value function, which is an estimate of the appropriateness of a control action given the observed state. In many applications, the value function that needs to be learned can be rather complex. It is then usual to use general function approximation methods, such as neural networks and fuzzy systems for approximating the value function. This approach has been the start of extensive research on fuzzy and neural reinforcement learning algorithms, and is the focus of this dissertation.

In most combinations of fuzzy systems and reinforcement learning, the environment is considered deterministic, where the rewards are known and the consequences of an action are well defined. In many problems, however, the consequence of an action may be uncertain or stochastic in nature. In that case, the agent deals with environments where the exact nature of the choices is unknown, or it is difficult to foresee the consequences or outcomes of events with certainty. Furthermore, an agent cannot simply assume what the world is like and take an action a ccording to tho se a ssumptions. Instead, it nee ds to consider multiple possible contingencies and their likelihood. In order to handle this key problem, instead of predicting how the system will respond to a certain action, a more appropriate approach is to predict a system probability of response (Berg et al., 2004).

In this dissertation, it is proposed a novel reinforcement learning approach that combines the universal function approximation capability of fuzzy systems and

probability theory. The proposed algorithm, seeks to exploit the advantages of both fuzzy inference systems and probabilistic theory to capture the probabilistic uncertainty of real world stochastic environments. This will allow the agent to choose an action based on a probabilistic distribution able to minimize negative outcomes (or maximize positive reinforcements) of future events.

The proposed generalized probabilistic fuzzy reinforcement learning (GPFRL) method is a modified version of the actor-critic learning architecture, where uncertainty handling is enhanced by the introduction of a probabilistic term into the actor and the critic learning. The addition of the probabilistic terms enables the actor to effectively define an input-output mapping by learning the probabilities of success of performing each of the possible output actions. In addition, the final output of the system is evaluated considering a weighted average of all possible actions and their probabilities.

The introduction of the probabilistic stage in the system adds robustness against uncertainties and allows the possibility of setting a level of acceptance for each action, providing flexibility to the system while incorporating the capability of supporting multiple outputs. In the present work, the transition function of the classic actor-critic is replaced by a probability distribution function. This is an important modification, which enables us to capture the uncertainty in the world, when the world is either complex or stochastic. By using a fuzzy set approach, the system is able to accept multiple continuous inputs and to generate continuous actions, rather than discrete actions as in traditional RL schemes. This dissertation will show that the proposed GPFRL is not only able to handle the uncertainty in the input states, but also has a superior performance in comparison with similar fuzzy-RL models.

1.2 Research Goal

The root of the proposed RL method is based on the introduction of probabilistic theory, in order to take advantage of its uncertainty handling capabilities, enabling agents to take decisions under the uncertainties inherent to stochastic environments, while keeping or improving the learning performance of similar existing methods.

Many of these past studies concentrated on increasing the learning speed and to test this, they assumed a predetermined kinematic structure, thus those studies were not generalized enough to be applied to other models and did not consider the uncertainty of real physical environments. The fundamental approach used here is to incorporate probabilistic theory to avoid deterministic actions, which, according to the literature review, is predominantly encouraging in order to achieve the best performance under uncertainty.

The development of this research pursued a bottom-up strategy, in which, the core functionalities of the typical actor-critic reinforcement learning method were created while introducing probabilistic theory, then, optimised through the gradient descent method, and finally tested in four different platforms.

1.3 Summary of Contributions

It will be seen later that the GPFRL framework is a modified version of the actor-critic technique. This by itself is not a new concept, but GPFRL is unique in several other aspects. The following contribution is derived from the work described in this dissertation:

A Generalized Probabilistic Fuzzy Reinforcement Learning method intended for continuous states and actions and is able to learn input-output mappings through

interactions with the environment. This method tackles 3 issues in control systems and decision making processes.

- Uncertainty in the outcome of actions is handled by a probabilistic approach.
- Learning from interaction, where system model is not readily available is handled by using reinforcement learning.
- Uncertainty in the inputs, which is handled by using a fuzzy logic control method.

The developed algorithm exhibits a comparatively fast learning speed and flexibility as it can be used for several different systems, where control or decision making under uncertainty is paramount. This concept has been tested through 4 different experiments:

- The random walk.
- The cart-pole balancing problem.
- A DC motor control.
- A real time experimental investigation of obstacle avoidance for a Khepera III mobile robot.

1.4 Outline of Dissertation

The logical structure and flow of the dissertation is summarised by Figure 1.1. The rest of this dissertation is organized as follows:

- **Chapter 2** Reviews some basic concepts on knowledge acquisition and machine learning defining the main advantages and difficulties on its use.

- **Chapter 3** Introduces the reinforcement learning problem and its biological origin, defining some commonly used terms within the reinforcement learning literature then, it presents a survey of different kind of reinforcement learning methods and analyses some commonly used algorithms for reinforcement learning. Finally it presents the use of reinforcement learning in the fields of decision making and control.

- **Chapter 4** Analyses the importance of uncertainty handling; reviews the concepts of fuzzy inference systems, then, it introduces the reader to probabilistic theory and its application to fuzzy logic. Finally, it presents an overview of fuzzy logic learning in general.

- **Chapter 5** Presents an overview of the complete GPFRL architecture and proposes a novel probabilistic reinforcement learning method.

- **Chapter 6** In this chapter, we perform some experiments employing the proposed GPFRL architecture. Four experiments are presented in this dissertation: a random walk on a 2-dimensional grid world, a control of a classic cart-pole balancing problem, GPFRL as a DC motor controller and finally the proposed GPFRL is implemented in a Khepera III mobile robot for obstacle avoidance.

- **Chapter 7** This chapter recalls the contributions of this work; it then states some observations and proposes some possible extensions and directions for future research.

Four appendices complement the chapters above as follows: first, appendix A contains the code used for the random walk example. Second, appendix B contains the code used for the cart-pole example. Appendix C contains the code used for the DC motor example. Finally, appendix D, contains the code used to operate the Khepera III mobile robot.

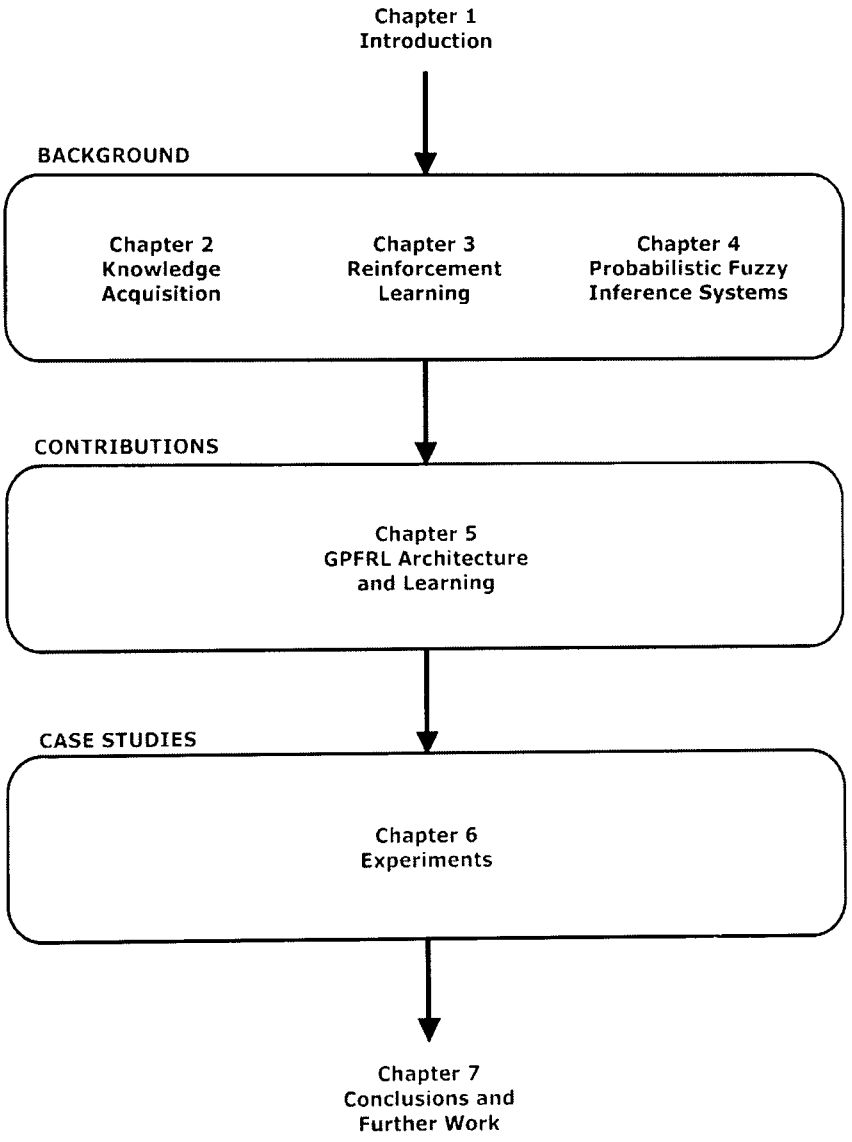

Figure 1.1: Flow of the dissertation.

Chapter 2

Knowledge Acquisition

2.1 Introduction

In its early beginnings, knowledge acquisition was seen as the transfer of expertise in problem solving from a human expert to a knowledge-based system. Since manual methods for this process were found to be very difficult and time consuming, knowledge acquisition was considered the bottleneck to building expert systems. To solve this problem, several different approaches like machine learning and data mining have been developed. This dissertation concerns an extension to a particular knowledge acquisition and knowledge based systems technique called reinforcement learning.

The rest of this chapter is organized as follows; section two will provide a review of the concepts of knowledge acquisition, section three presents an in-depth review of machine learning concepts and some of the most well-known algorithms used for machine learning, and finally section four and five presents the conclusions and summary of the present chapter. The aim of this chapter is to

situate the approach in this dissertation rather than provide a detailed review of the very wide range of research that falls under knowledge acquisition field.

2.2 What Is Knowledge Acquisition?

In computer science knowledge acquisition can be defined as the transfer and transformation of potential problem solving expertise from some knowledge source to a program (Kohonen, 1988).

Knowledge acquisition has been recognised as a bottleneck in the development of knowledge based systems due to the difficulty in capturing the knowledge from the experts; that is externalising and making explicit the tacit knowledge, normally implicit inside the expert's head.

Knowledge acquisition techniques can be grouped into three categories: manual, semi-automated (interactive) and automated (machine learning and data mining). For the manual and semi-automated category, the expert must only answer simple questions (cases); so that the more data is labelled by the expert, the better the obtained results (Liu and Li, 2005). However, one problem of this approach is that we do not know in advance how many cases we will need to ask in order to get an accurate model and asking thousands of cases to the expert is usually impractical. Therefore, an automatic way to infer an appropriate behaviour is highly desired.

Automated methods provide an automatic way to learn behaviours from data (machine learning) or to extract patterns from it (data mining). Data mining is a branch of computer science and artificial intelligence. It can be defined as the process of extracting patterns from data and it can be seen as an increasingly important tool by modern business to transform data into business intelligence giving an informational advantage. It is currently used in a wide range of profiling

practices, such as marketing, surveillance, fraud detection, and scientific discovery.

The main difference between these two methods is that data mining can be defined as a process of applying several different methods like neural networks, clustering, genetic algorithms, decision trees and support vector machines to data with the intention of uncovering hidden patterns (Kantardzie, 2002), whilst machine learning is the study of computer algorithms that improve automatically through experience (Mitchell, 1997). Figure 2.1 shows the taxonomy of knowledge acquisition and its link to some of the most common reinforcement learning methods. The darker blocks mark the categories that contain the algorithm proposed in this work.

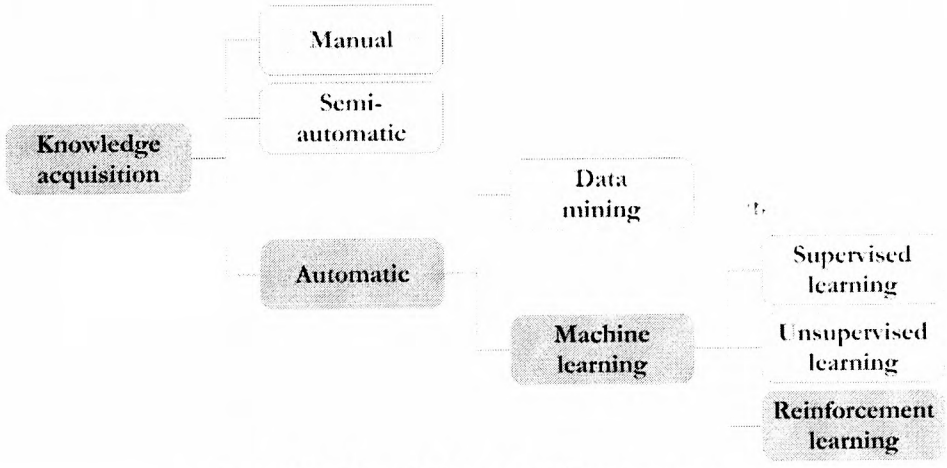

Figure 2.1: Knowledge acquisition taxonomy.

2.3 Machine Learning

Learning, like intelligence, covers su ch a broad range of processes that it is difficult to define precisely, such as knowledge acquisition, understanding and skill acquisition by study, instruction or experience and even modification of a behavioural tendency by experience. Before describing machine learning, we

should be clear what is meant by learning. In (Hebb, 1949) learning is defined as the ability to perform new tasks or perform old tasks better (faster, more accurately, etc.) as a result of changes produced by the learning process. Furthermore, when this learning is achieved by a machine, rather than a person, it is called machine learning.

This learning capability is an inherent characteristic of human beings, which enable us to acquire "ability" or "expertise" so we can improve our performance while executing repetitive tasks. Similarly, machine learning usually refers to the changes in systems that perform certain tasks. Such tasks can involve recognition, diagnosis, planning, robot control, prediction, etc. These system changes can be either enhancements to already performing systems or synthesis of new systems. It can be said then, that a machine has learned whenever its structure, program, or data has been changed, based on its inputs and/or in response to external information, so that there is an improvement in its expected future performance. In other words, machine learning involves developing computer systems that automatically improve their performance over a specific task, through experience.

Now that we have defined what machine learning is, the next question to ask is, why should machines have to learn? There are several reasons why machine learning is important. An important reason, especially for the fields of human psychology is that the achievement of learning in machines might help us understand how animals and humans learn. But for the engineering fields some of the reasons presented by (An et al., 1988) are:

- Some tasks cannot be defined well except by example; that is, we might be able to specify input/output pairs but not a concise relationship between inputs and desired outputs. Machine learning methods can be used to adjust their internal structure to produce correct outputs for a large number of sample inputs.

- It is possible that hidden among large piles of data are important relationships and correlations. Machine learning methods can often be used to extract these relationships.

- Some machines might not work well when certain characteristics of the working environment are not completely known at design time. Machine learning methods can be used for on-the-job improvement of existing machine designs.

- The amount of knowledge available about certain tasks might be too large for explicit encoding by humans. Machines that learn this knowledge gradually might be able to capture more of it than humans would want to write down.

- Under stochastic environments, machines that can adapt to a changing environment would reduce the need for constant redesign.

- New knowledge about tasks is constantly being discovered by humans. Continuing redesign of AI systems to conform to new knowledge is impractical, but machine learning methods might be able to track much of it.

It is important to highlight that, the techniques of machine learning do not interact directly with a human expert, but instead with data descriptions of the problem to be solved. For example; supervised learning, the most common machine learning paradigm, involves presenting a computer program with a "training set" of example problem descriptions, together with the known solution in each case. The machine learning program is then able to infer a pattern to the example/solution pairs such that it is afterwards able to solve problems which it has not seen before.

In the next two subsections, two different ways to classify machine learning algorithms are presented.

2.3.1 Learning-Based Classification

There are many ways to classify the machine learning paradigms. (Langley, 1995) classifies the learning paradigms according to the way they learn in five categories: inductive learning (e.g., acquiring concepts from sets of positive and negative examples); instance-based; analytical learning (e.g., explanation-based learning and certain forms of analogical and case-based learning); connectionist learning methods and evolutionary algorithms.

2.3.1.1 Inductive Learning

Inductive learning seeks to find a general description of a concept from a set of known examples and counterexamples. It is assumed that the known examples have to be representative of the whole space of possibilities. Given an encoding of the known background knowledge and a set of examples represented as a logical database of facts, an inductive learning system will derive a hypothesised logic program which entails all the positive and none of the negative examples. The resulting description distinguishes not only between the known examples of the concept (a recall task), but can also be used to make predictions about unknown examples.

Inductive learning employs condition-action rules, decision trees, or similar logical knowledge structures, where the information about classes or predictions is stored in the rule actions sides of the rules or the leaves of the tree. Learning algorithms in the rule-induction framework usually carry out a greedy search through the space of decision trees or rule sets, using statistical evaluation functions to select attributes to incorporate into the knowledge structure.

Some examples of inductive learning include:

- Approximate identification (PAC-learning) (Valiant, 1984).
- Boosting (Schapire, 1990).
- TDIDT (Top-Down Induction of Decision Trees) (Quinlan, 1986).

2.3.1.2 Instance-Based Learning

Represents knowledge in terms of specific cases or experiences and relies on flexible matching methods to retrieve these cases and apply them to new situations. Instead of performing explicit generalization, it compares new problem instances with instances seen in training, which have been stored in memory. Some examples of Instance-based learning include:

- K-nearest neighbor algorithm (Fix and Hodges, 1951).
- Locally weighted regression (Cleveland, 1979).

2.3.1.3 Analytic Learning

Analytic learning represents knowledge as rules in a logical form (as inference rules) usually by employing a performance system that uses search to solve multi-step problems. This knowledge is acquired by applying background knowledge (from rigorous expert explanations) to very few examples (often only one). Some issues with analytic learning are:

- The background knowledge and complex structure necessary to store explanations make analytic learning systems infeasible for large-scale information retrieval.

- Analytic learning systems are domain-knowledge intensive, thus requiring a complete and correct problem-space description.

One of the best well-known examples of analytic learning is:

- Explanation-based learners (EBL) (Mitchell et al., 1986).

2.3.1.4 Connectionist Learning

In connectionist learning the knowledge is represented as a multi-layer network of threshold units that spreads activation from input nodes through internal units to output nodes. Many proponents of connectionist learning believe that this form of learning is the nearest to that of the human brain. In connectionist learning, the weights on the links determine how much activation is passed on in each case. The activations of output nodes can be translated into numeric predictions or discrete decision about the class of the input. This connectionist paradigm aims at massively parallel models that consist of a large number of simple and uniform processing elements interconnected with extensive links, that is, artificial neural networks and their various generalisations. In many connectionists' models, representations are distributed through a large number of processing elements. This parallel nature makes connectionist learning algorithms good at flexible and robust processing. The most well-known algorithm of this group is:

- Back propagation (Rumelhart et al., 1986).

The back propagation algorithm uses two-layer networks in which every node in a layer receives an input from every node in the preceding layer. The term back propagation refers to the procedure for updating weights across the layers. Starting from the final output, the difference between the actual and desired output (the error) is divided and allocated to the directly contributing nodes. These

nodes, in turn, do the same to their inputs, with the resulting effect that weight changes are propagated backwards through the network.

2.3.1.5 Evolutionary Learning

Evolutionary learning is defined as any kind of learning task which employs as its search engine a technique belonging to the evolutionary computation field (Michalewicz, 1996). Evolutionary learning is a set of optimisation tools inspired by natural evolution, were a population of candidate solutions (individuals) are transformed (evolved) through a certain number of iterations of a cycle, containing an almost blind recombination of the information contained in the individuals and a selection stage that directs the search towards the individuals considered good by a given evaluation function. In this scheme, the characteristics of an individual which is fit (and therefore good at competing for resources) are more likely to be passed to the next generation than the characteristics of an unfit individual.

One of the most representative algorithms of this group is generic algorithms (GA). Genetic algorithms attempt to find solutions to problems by mimicking this evolutionary process (Holland, 1992). GAs supports a population of individuals, each one representing a possible solution to the problem of interest. Although proponents of genetic algorithms recognise that the approach does not always deliver the optimal solution to a problem, they argue that the approach can deliver a "good" solution "acceptably" quickly.

Some evolutionary learning techniques include:

- Genetic algorithms (Holland, 1992).
- Genetic programming (Barricelli, 1954).
- Evolutionary programming (Fogel et al., 1966).

2.3.2 Problem-Based Classification

A better classification, using a more general problem-based point of view suggests three main categories, supervised learning, unsupervised learning and reinforcement learning.

2.3.2.1 Supervised Learning

Supervised learning is a machine learning technique for function approximation obtained from training data. A supervised learning problem is one where we are supplied with data as a set of input/output pairs, assumed to have been sampled from an unknown function and we wish to reconstruct the function that generates the data.

Supervised learning networks learn by being presented with pre-classified training data (i.e. input and target output pairs). This type of learning requires a "trainer" or "supervisor", who supplies the input-output training pairs. The learning agent adapts its parameters using some algorithms in order to generate the desired output patterns from a given input pattern and it achieves this by generalizing from the presented data to unseen situations in a "reasonable" way. This is equivalent to having a supervisor who can tell the agent how the output should have been, so that the learning agent can learn from the "example". Most supervised learning techniques attempt to increase the accuracy of their function approximation through generalization: for "similar" inputs, the outputs are assumed "similar".

In general, the goal of the supervised learning is to estimate a function $g(x)$, given a set of points $(x, g(x))$. The basic problem of supervised learning deals with predicting the response variables from the independent variables. In essence, the input X is a collection of p associated variables, and for each X, an

observed value Y, of the output, is the supervisor. The goal is to train the learner, using the training set based on N samples of the pair (Y,X), so that it can predict the value $\hat{Y}(x)$ from a future observation x. The output of the function can be a continuous value (regression), or can predict a class of the input object (classification). Figure 2.2 shows a basic scheme of general supervised learning algorithms.

The four more important issues to consider in supervised learning are:

- **Bias-variance trade-off.** The performance of an estimator θ^* of a parameter θ is measured by its mean square error (MSE) that is shown to be: $MSE = Var(\theta^*) + Bias(\theta^*)^2$. Although the lack of bias is an attractive feature of an estimator, it does not guarantee the lowest possible value of the mean square error. This minimum value is attained when a proper trade-off is found between the bias of the estimator, and its variance. So as to make the value of the above expression smallest. As a matter of fact, it is commonly observed that introducing a certain amount of bias into an otherwise unbiased estimator can lead to a significant reduction of its variance, so much so that the MSE will be reduced and therefore the performance of the estimator will be improved.

- **Function complexity and amount of training data.** If the true function is simple, then an "inflexible" learning algorithm with high bias and low variance will be able to learn it from a small amount of data. But if the true function is highly complex, then the function will only be learnable from a very large amount of

training data and by using a "flexible" learning algorithm with low bias and high variance.

- **Dimensionality of the input space.** If the input states vectors have are of a large dimension, the learning problem can be difficult even if the true function only depends on a small number of those features. This is because the many "extra" dimensions can confuse the learning algorithm and cause it to have high variance. Hence, high input dimensionality typically requires tuning the classifier to have low variance and high bias.

- **Noise in the output values.** If the desired output values are often incorrect, then the learning algorithm should not attempt to find a function that exactly matches the training examples. This is another case where it is usually best to employ a high bias, low variance classifier.

The proposed GPFRL method tackles all this shortcomings in several different ways. The model bias is one of the main reasons why reinforcement learning algorithms often need so many trials to successfully learn a task. Although there is no general solution to this problem this problem can be tackled by learning probabilistic models in order to reduce model bias, by incorporating the model's uncertainty into planning and policy learning as has been proposed by (Deisenroth and Rasmussen, 2010) who proposed that in order to reduce model bias, it is required a probabilistic model to faithfully describe the uncertainty of the model in regions of the state space that have not been encountered before. Moreover, this model uncertainty is required to be taken into account during the policy evaluation. (Deisenroth and Rasmussen, 2010) implemented the probabilistic model using a Gaussian process, which also allows for closed form approximate inference for propagating uncertainty over longer horizons.

The function complexity and amount of training data issue is a non important issue in reinforcement learning methods as training data is not required. Reinforcement learning algorithms can be considered "flexible" as they are able to learn complex true functions with low bias and high variance.

To enhance the generalization ability of average reward reinforcement in continuous state space, this dissertation proposes an improved algorithm based on fuzzy inference system. In reinforcement learning, agent accesses to knowledge through the interaction with the environment. The reward signal from the environment carries out comprehensive appraisal of the effect of the action, which is operated by the agent. However, when the learning environment is complex and continuous, it will increase the difficulty of learning process and reduce the learning efficiency. In response to these disadvantages, we use fuzzy inference system as the approximator to generalize the continuous state space.

A final issue is whether the environment is noisy or deterministic. Noise may appear, for example, in the rules which govern transitions from state to state, and in the final reward signal given at the terminal states. Fuzzy inference systems are intrinsically able to reduce the effect of the noise in the input and output values as it is discussed in (Saade, 2011). Most of the ordinary adaptive controllers are based on a special noise prediction and estimation, and if the nature of the noise changes or its statistical characteristics altered then these controllers will have poor performance, while on-line learning controllers, such as the one proposed in this dissertation, can self-adapt themselves to such changes and still maintain high performance.

Over the last decade, supervised learning has been the focus of research of many computer scientists, with several different methods developed. Some of the most remarkable of these methods are back propagation, artificial neural networks, logistic regression, naïve Bayes and decision trees within others.

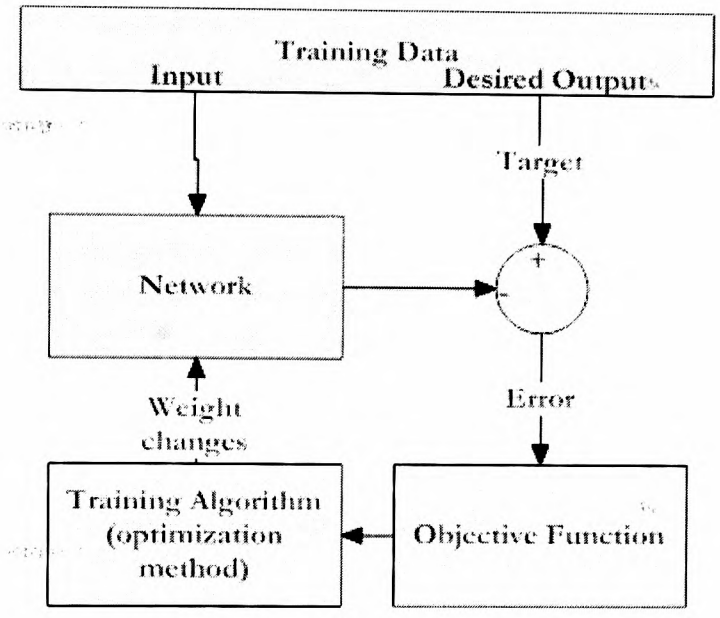

Figure 2.2: Supervised learning scheme.

2.3.2.2 Unsupervised Learning

In absence of supervisors, the desired output for a given input instance is unknown; therefore the supervisor has to adapt its parameters by itself. In this case, such type of learning is called "unsupervised learning". In unsupervised learning, the goal is harder because there are no pre-determined categorizations.

In contrast with supervised learning, unsupervised learning is focused to solve a class of problems in which the goal is to determine how the data is organized or classified. In unsupervised learning the learning agent is presented with a training set of vectors without function values so the problem consists in partitioning this training set into subsets. For an unsupervised learning rule, the training set consists of input training patterns only. Therefore, the learning agent is trained without using any supervisor, so that the learning agent learns to adapt based on the experiences collected from the previous input training patterns.

If we consider a learning agent an input sequence $x_1, x_2, x_3, \ldots$, where x_t is the sensory input at time t. This input, or data, could correspond to any kind of sensory input, an image from a camera, a sound, etc. The machine simply receives inputs $x_1, x_2, x_3, \ldots$ but obtains neither supervised target outputs, nor rewards from its environment. Despite that the machine never receives any feedback from its environment, it is possible to develop a formal framework for unsupervised learning based on the notion that the machine's goal is to build representations of the input that can be used for decision making, predicting future inputs, efficiently communicating the inputs to another machine, etc.

Some unsupervised learning issues include:

- Objective function not as obvious as in supervised learning. Usually try to maximize likelihood (measure of data compression).
- Local minima (non convex objective).
- Uses inference as subroutine (can be slow – no worse than discriminative learning).

Some of the forms of unsupervised learning methods include clustering which is the assignment of a set of observations into subsets (clusters) so that observations in the same cluster are similar in some way and blind source separation based on Independent Component Analysis (ICA) (Comon, 1994).

Some examples of unsupervised learning algorithms include:

- A Kohonen map, or self-organizing feature map (Kohonen, 1988).
- Hebbian learning (Hebb, 1949).

2.3.2.3 Reinforcement Learning

In reinforcement learning, the learning agent does not explicitly know the input-output instances; instead it receives feedback from its environment. This feedback signal helps the learning agent to decide whether the selected action will receive a reward or a punishment based on its performance. The learning agent thus adapts its parameters based on the rewards or punishments of its actions.

Reinforcement learning is a current and important topic of research, where efforts are placed in the design and study of RL algorithms that can make an efficient use of the available information while having good computational scalability.

Reinforcement learning differs from supervised learning in two important aspects; First, supervised learning is learning from examples provided by a knowledgeable external supervisor, whilst reinforcement learning is defined not by characterizing learning algorithms, but by characterizing a learning problem (Sutton and Barto, 1998) so that the agent never sees examples of correct or incorrect behaviour. Instead, it receives only positive and negative rewards for the actions it tries. Supervised learning is an important kind of learning, but alone it is not adequate for learning from interaction. In interactive problems it is often impractical to obtain examples of desired behaviour that are both correct and representative of all the situations in which the agent has to act. In some cases, like uncharted territories, an agent must be able to learn from its own experience. Second, the exploration-exploitation dilemma is of important consideration in reinforcement learning whilst the entire issue of balancing exploration and exploitation does not even arise in supervised learning.

When compared to unsupervised learning, the main difference is that in unsupervised learning the learner receives no feedback from the world at all, whilst the reinforcement learning agent receives information regarding the success or failure of the actions taken. In addition, unsupervised learning agents, learn to

differentiate the input data, in order to classify it, in reinforcement learning, the agents learn input-output mappings through repeated interactions with the environment.

2.4 Conclusions

The concept of knowledge is still a not well defined one. Research has yet to conclude the underlying mechanism in the human or animal brain that enables us to learn. Learning has been defined as the process of acquiring a skill or knowledge in the case of humans. The concept of learning itself is no different if applied to a machine. Many learning methods of machine learning are inspired in biological mechanisms of our brains.

Problem-based classification of machine learning algorithms can be further divided into tree subcategories. Supervised learning is basically learning from examples, where sets of input-output pairs are presented to the learner, which then through some algorithm like the back propagation algorithm, a function is approximated in order to generalize future inputs. In unsupervised learning, the learning agent doesn't receive any feedback whatsoever from the environment, yet, it is able to recognize features of the input sets, and classify it. Finally, in reinforcement learning, the learning agent receives sparse information from the environment, regarding a state of success or failure as a result of a previous action.

Reinforcement learning can overcome some important limitations of supervised learning present in cases where "example sets" are not readily available, instead it uses a reinforcement signal in order to find an optimal policy either for a decision making or a control action. Learning by reinforcement is much more difficult than supervised learning. RL systems must learn from limited or poor information

(reinforcement signal) in contrast with supervised learning, where the information consists on a complete set of input-output examples. In many cases the information required to create input-output sets for training might not be available/accessible.

2.5 Summary

This chapter presented a brief review on knowledge acquisition, its definition and classification, highlighting and precisely locating the scope of this work. It was presented two types of classifications for knowledge acquisition; a learning based classification and a problem-based classification. Additionally a brief overview of different learning methods has been provided. The next chapter will describe in more detail the reinforcement learning problem, and will provide an introduction to reinforcement learning and several different reinforcement methods.

Chapter 3

Reinforcement Learning

3.1 Introduction

Reinforcement learning is an important aspect of much learning in most animal species. In the context of machine learning, reinforcement learning refers to a set of different methods oriented to solve a specific group of problems. Such problems consists on determining or approximating an optimal policy through repeated interactions with the environment (i.e., based on a sample of experiences of the form state–action–next state–reward). In many cases, like in uncharted scenarios, an agent will benefit the most in learning from its own experience (Sutton and Barto, 1998).

The rest of this chapter is organized as follows; section two will explain how RL is produced in the human brain, section three provides a brief review of the most commonly used RL terms that will be used through all the rest of this dissertation. Section four presents RL taxonomy, sections five to seven will provide an overview of the classification of RL methods. In section eight, a more detailed

review of some of the most well-known methods is presented. Sections nine and ten describe the use of RL methods for decision making and control and finally, section eleven summarizes the contents of this chapter.

3.2 Reinforcement Learning and the Brain

In nature it is observable that humans and animals can learn novel behaviours under a wide variety of environments; the animal brain seems to have certain mechanisms that enable us to learn behaviours based on past experiences (Blackmore, 2006). In psychological theory, reinforcement is a term used in operant conditioning and behaviour analysis for the delivery of a stimulus (immediately or shortly) after a response, that results in an increase or decrease in the probability of that response. Recently reinforcement learning models have been applied to a wide range of neurobiological and behavioural data. Specifically the computational function of neuro-modulators such as dopamine, acetylcholine, and serotonin have been addressed using the reinforcement learning framework (Niv, 2009). More specifically, it has being proposed by (Doya, 2002) that:

- Dopamine signals reward prediction error.
- Serotonin controls the time scale of prediction of future rewards.
- Noradrenalin controls the width of exploration.
- Acetylcholine controls the rate of memory updates.

From all these mentioned neuro-modulators, dopamine is the most studied, perhaps due to its implication in certain brain conditions such as Parkinson's disease, schizophrenia, and drug addiction as well as its function in reward learning and working memory. The link between dopamine and RL was made in the mid '90s based on a hypothesis that viewed dopamine as the brain's reward

signal. Recent research (Schultz et al., 1997) discovered that the firing rate of dopamine neurons in the *ventral tegmental* area and *substantia nigra* (Figure 3.1), appear to mimic the error function in the temporal difference algorithm (section 3.6.1).

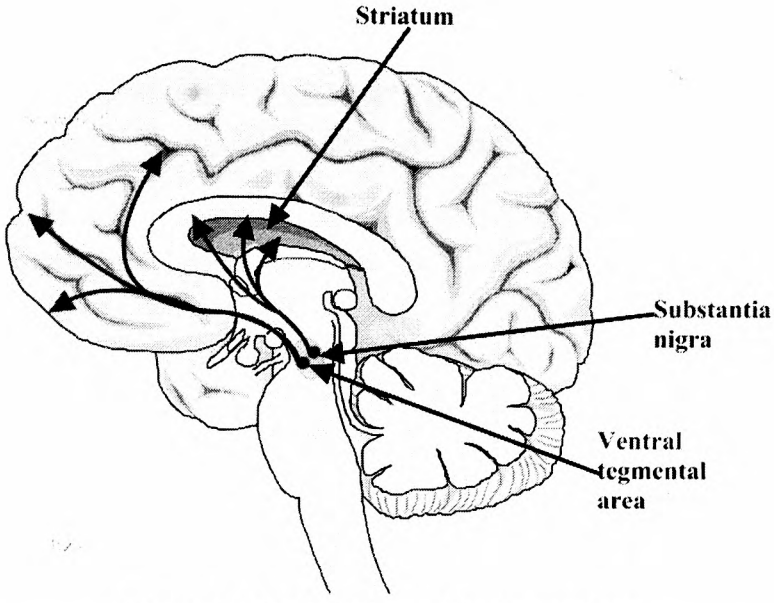

Figure 3.1: Sagital view of the human brain.

When the amount of actual reward is larger than animal's expectation of reward this dopamine neurons fire in the midbrain, therefore it appears that dopamine neurons can be considered to code the reward prediction error. In the reinforcement learning algorithm, the reward prediction error has an essential role to learn the optimal behaviour. Therefore, its being hypothesized that a reinforcement learning algorithm is implemented in the basal-cortico circuit of the brain (Doya, 2002).

Despite that some systems, like reinforcement learning as outlined above, can be used to describe many phenomena found in animal behaviour, the question remains whether all known modes of animal behaviour can be attributed to

systems based on relatively simple rules, or whether other structures are required for higher (intelligent) processes, for example: behaviours that rely on reflecting, or reasoning.

3.3 Common Terms

The next subsections describe the most important terms used in reinforcement learning that will be use through the entire dissertation.

3.3.1 The Agent

A system moves and/or interacts with the environment through specific actions. The agent observes the state of the environment, and uses a learned policy in order to select an appropriate action (output). The agent also must have a goal or goals relating to the state of the environment. For example, an agent can be a robot moving over the floor, or a manipulator trying to reach for a specific part.

3.3.2 The Policy

The policy (π) is a decision rule that dictates what action to take at every possible state. The goal of the learning agent is to find a policy that maximizes the total expected reward (or the total discounted rewards) that it will receive over time, known as "the expected return". For example: for the present dissertation, given any particular state, the use of the policy will determine a probability distribution for the agent's future rewards, or as we will call it, a probability of success.

3.3.3 The Environment

The environment is the external and immediate world to the agent. The environment interacts with the agent, dynamically affecting its state. It can be represented in many different ways. For example: the environment for a robot can be composed by the walls surrounding it or the friction of its wheels with the floor, or the floor inclination; the environment for a system like the cart-pole (described in Chapter 6) is composed by the gravity forces acting on the mass.

Agent environments are classified based on different properties that can affect the complexity of the agent decision-making process (Russell and Norvig, 1994).

- **Accessible or inaccessible:** An accessible environment is one in which the agent can obtain complete, timely and accurate information about the state of the environment.

- **Deterministic or non-deterministic:** In a non-deterministic environment, the state that will result from a n action is not guaranteed even when the system is in a similar state. This uncertainty presents a greater challenge to the agent designer than deterministic systems.

- **Static or dynamic:** Static environments remain unchanged except for the results produced by the actions of the agent. Whilst dynamic environments are created when other processes operate in them, thereby changing the environment outside the control of the agent.

- **Discrete or continuous:** An environment is discrete if there are a fixed, finite number of actions and percepts in it.

3.3.4 The Reward

The reward function is a signal that expresses the failure or success of performing a specific action and it is extracted from a state observation; in other words the reward maps each perceived state (or a state-action pair) of the environment to a single number. The reward indicates the intrinsic desirability of the observed state; therefore, if a policy selected action returns a negative reward, then the policy may be changed in order to select a different action for that given state in the future. For example, a simple system like the cart-pole can be credited with a reward of zero for every action that keeps it in balance and punished with a reward of -1 as soon as the controller takes an action that leads the pole out of balance. In the case of the grid world problem, where we want to use a value based reinforcement learning algorithm to learn how to reach the goal state as quickly as possible, a natural way to model goal states in a reward function is to give a positive reward on reaching the goal state and zero reward on all other steps. In the grid world this would translate to a positive reward for reaching the goal state and zero reward on each other step. However, this reward function would then require a discount factor strictly lower than one. Otherwise, any action that eventually exits the grid world is optimal and there is no incentive to reach the goal quickly. An alternative method would be to set the reward for each transition to some negative value and to use this as an incentive to find the goal quickly.

3.4 Reinforcement Learning Taxonomy

Being a very flexible learning technique, to date, several methods and algorithms have been developed. Each of these can be classified under one or more categories according to different criteria. There are mainly three basic criteria to classify reinforcement learning methods:

- By the presence of a system model.
- By its structure.
- By the way they learn and select a correct policy.

Figure 3.2 presents a chart showing the reinforcement learning taxonomy, along with some well-known algorithm examples. In Figure 3.2 the shaded blocks show the model that forms the principal focus of this dissertation. Figure 3 .2 also present the classification of some well-known reinforcement learning methods.

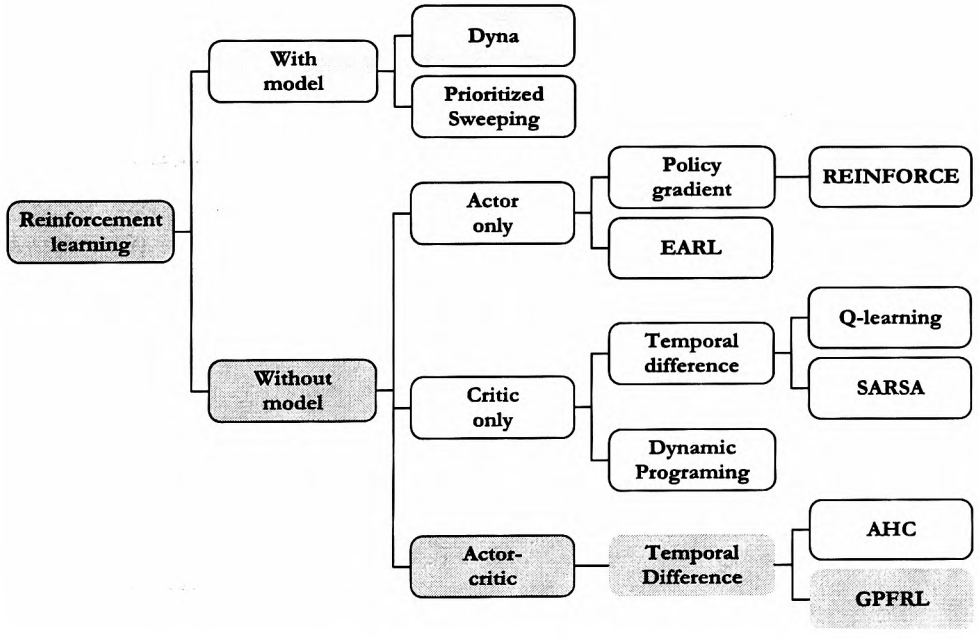

Figure 3.2: Reinforcement learning taxonomy.

3.5 Model-Based Methods

Model-based or indirect methods act in two phases: first, they learn the transition probability and reward functions and second, they make use of those transition probabilities in order to compute the Q function by means of, for example, the

Bellman equations. Some of the best well know algorithms in this category are: Dyna (Roy et al., 2005) and prioritized sweeping (Demspster, 1969, Moore and Atkeson, 1993).

3.6 Model-Free Methods

Model free or direct methods learn the policy by directly approximating the Q-function with updates from direct experience. These methods are sometimes referred to as the Q-learning family of algorithms. See (Sutton, 1988) or (Watkins, 1989). Some of the best well know algorithms in this category are: adaptive heuristic critic (AHC) learning (Barto et al., 1983), Q-learning (Watkins, 1989) and SARSA (Rummery and Niranjan, 1994).

Model free methods can be further classified in three sub groups depending on whether the algorithm focuses on learning the policy or the value function, critic only, actor only, and actor-critic (Barto et al., 1983).

3.6.1 Critic Only Methods

Also called value function-based methods, are based on the idea to find the optimal value function and then to derive an optimal policy from this value function. Some of the most important learning algorithms in RL are critic-only methods, such as:

- Dynamic programming (Bellman, 1957).
- Temporal Difference (Sutton and Barto, 1987, Sutton, 1988).

3.6.2 Actor Only Methods

In contrast with critic only methods, actor only methods learn the policy, which is a function $\pi(s)$ that depends only on the current state and therefore a value function is never defined.

In actor only methods, the learning agent uses an explicit representation of its behaviour with the goal of improving it by searching the space of possible policies **P**. Therefore, an actor only method will be feasible, if its search space is restricted to a subset of **P**. In so doing, a large portion of prior knowledge can be integrated into the learning task and hence, the search complexity be reduced. However, in some cases, incorporating large amounts of background knowledge is hard to accomplish, e.g. whe n the task is not fully understood making it difficult to specify an appropriate subclass of policies.

Some important actor only algorithms include:

- Associative Reinforcement Learning Algorithms (Barto and Anandan, 1985).
- Policy gradient algorithms (Cheeseman, 1985).
    - REINFORCE (Williams, 1992).
- EARL, evolutionary algorithm RL (Moriarty et al., 1999).

3.6.3 Actor-Critic

As proved by (Kalyanakrishnan and Stone, 2009) both previously described methods have strong theoretical foundations, however, their performance cannot be a ssured f or the c ases when the agent has to cope with deficient function

approximation and partial observability. Critic only methods have a superior sample complexity and asymptotic performance when provided complete state information; yet, critic only methods are robust to inadequate function approximation and noisy state information.

Although value function-based methods and actor-only methods are contrasting approaches to solve reinforcement learning tasks, it is possible to combine their advantages. Actor-critic (AC) methods are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function. The policy structure is known as the *actor*, because it is used to select actions, and the estimated value function is known as the *critic*, because it criticizes the actions made by the actor. When combined into an AC structure, the learning is always on-policy: the critic must learn about and critique whatever policy is currently being followed by the actor. The critique takes the form of a TD error (a scalar signal) and represents the sole output of the critic which drives all learning in both actor and critic, as seen in Figure 3.3.

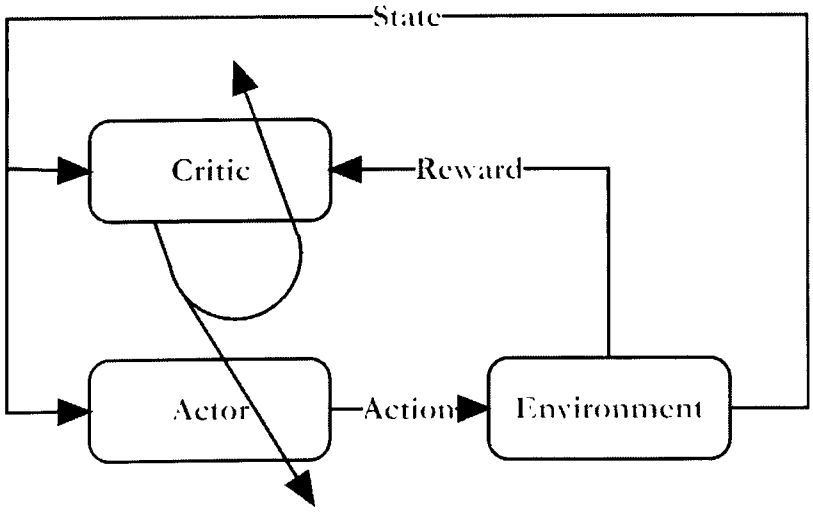

Figure 3.3: Actor-critic architecture.

In an AC system at any given time, the critic is learning the values for the Markov chain that comes from following the current policy of the actor. The actor is constantly learning the policy that is greedy with the respect to the critic's current values.

AC methods were among the first reinforcement learning algorithms to use temporal-difference learning. These methods were first studied in the context of a classical conditioning model in animal learning by (Sutton and Barto, 1981). Later, (Barto et al., 1983) successfully applied AC methods to the cart-pole balancing problem, where they defined for the first time the terms actor and critic.

In the simplest case of finite-state and action spaces, the following AC algorithm has been suggested by (Sutton and Barto, 1998). After choosing the action a_t in the state s_t and receiving the reward r_t, the critic evaluates the new state and computes the temporal-difference (TD) error, δ_t, according to (3.1):

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{3.1}$$

where γ is the discounting rate and V is the current value function implemented by the critic. After that, the critic updates its value function:

$$V(s_t) \leftarrow V(s_t) + \alpha_t \delta_t \tag{3.2}$$

where α_t is the critic's learning rate at time t. The key step in this algorithm is the update of actor's parameters. If TD error is positive, the probability of selecting action $a = a_t$ in the state $s = s_t$ in the future should be increased since action a has resulted in a better than expected state value. By reverse logic, the probability of selecting a_t in the state s_t in the future should be decreased if the

TD error is negative. Suppose the actor chooses actions stochastically using the Gibbs softmax method:

$$\Pr\{a_t = a \mid s_t = s\} = \frac{e^{\theta(s,a)}}{\sum_b e^{\theta(s,b)}} \tag{3.3}$$

where $\theta(s,a)$ is the value of the actor's parameter indicating the tendency of choosing action a in state s. Then, these parameters are updated as follows:

$$\theta(s_t,a_t) \leftarrow \theta(s_t,a_t) + \beta_t \delta_t \tag{3.4}$$

where β_t is the actor's learning rate at time t. A more detailed description of the temporal difference algorithm will be given in chapter 3.8.1.

An AC system first updates the value in every state once, then it updates the policy in every state once, finally the process is repeated, this occurs in the form of incremental value iterations. This process is a form of dynamic programming, which is guaranteed to converge to the optimal policy. If it instead updates all the values repeatedly in all the states until the values converge, then updates all the policies once, then repeats, then it reduces to policy iteration, another form of dynamic programming with guaranteed convergence. If it updates all the values N times between updating the policies, then it reduces to modified policy iteration, which is also guaranteed to converge to optimality.

It would seem that an actor-critic system with a lookup table is guaranteed to converge to optimality no matter what. Surprisingly, that is not the case. Although it always converges for $\gamma < 0.5$ (Williams & Baird 1993), it does not always converge for larger γ, as shown by (Baird, 1999).

TABLE 3.1 ACTOR-CRITIC ALGORITHM

Algorithm

Input: States $s \in S$, Actions $a \in A(s)$, Initialize α, γ.
Output: Policy $\pi(s,a)$ responsible for selecting action a in state s.
for (all $s \in S$, $a \in A(s)$) do
  $p(s,a) \leftarrow 0$;
  $\pi(s,a) \leftarrow \frac{e^{p(s,a)}}{\sum_{b=1} |A(s)| e^{p(s,a)}}$;
end
while True do
  Initialize s;
  for ($t=0$; $t<T_m$; $t=t+1$) do
    Choose a from s using $\pi(s,a)$;
    Take action a, observe r, s';
    $\delta = r + \gamma V(s') - V(s)$;
    $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$;
    $p(s,a) \leftarrow p(s,a) + \beta\delta$;
    $\pi(s,a) \leftarrow \frac{e^{p(s,a)}}{\sum_{b=1} |A(s)| e^{p(s,a)}}$;
    $s \leftarrow s'$;
  end
end

We conclude listing some of the most important advantages of using actor-critic methods:

- Explicit representation of policy as well as value function.
- Minimal computation to select actions.
- Can learn an explicit stochastic policy.

- Can put constraints on policies.
- Appealing as psychological and neural models.

Table 3.1presents a pseudo code example for the actor critic learning method. The best well known example of an actor-critic algorithm is the adaptive heuristic critic algorithm (AHC) (Barto et al., 1983) where this concept was introduced for the first time.

3.7 On-Policy / Off-Policy Methods

In either of the model-based or the model-free methods there are two basic strategies to update the policy's value function, on-policy learning and off-policy learning.

Off-policy methods can update the estimated value functions using hypothetical actions, those that have not actually been tried. Again, the behaviour policy is usually "soft" (includes an element of exploration). As an added advantage, off-policy algorithms can separate exploration from control, whilst on-policy algorithms cannot. A well know off-policy learning method is Q-Learning (Watkins, 1989) which is described in more detail in section 3.8.2.

In contrast with off-policy methods, on-policy methods learn the value of the policy that is used to make decisions using results from executing the actions determined by some policy, so that the value functions are updated based strictly on experience. These policies are usually non-deterministic and include an element of exploration to the policy. An example of an on-policy learning method is the SARSA algorithm (Rummery and Niranjan, 1994) will be described in more detail in section 3.8.3.

3.8 Reinforcement Learning Algorithms

3.8.1 Temporal Difference

As with other artificial learning algorithms, TD methods (Sutton, 1988) have been the subject of study of neuroscience. In research studies it has been discovered that the firing rate of dopamine neurons in two key areas of the brain, the ventral tegmental area (VTA) and substantia nigra (SNc) (see Figure 3.1), appear to mimic the error function as described by (Schultz et al., 1997). This error function reports the difference between the estimated reward at any given state or time step and the actual reward received, so that the larger the error function, the larger the difference between the expected and actual reward. When this is paired with a stimulus that accurately reflects a future reward, the error can be used to associate the stimulus with the future reward. Several experiments using brain scanners have dopamine cells appear to behave in a similar manner.

Temporal-difference (TD) methods were formalized and studied by (Sutton, 1988) as a solution to the problem of making multi-step predictions of future events based on past experience. Before Sutton's formalization, well-understood techniques for learning predictions were trained using differences between predictions and the actual future outcomes. With TD methods, learning was applied using the differences between temporally successive predictions.

The example used by Sutton is that of a weatherman making a prediction on Monday about if it will rain on Saturday. The conventional approach would have been to wait until Saturday, observe if it rained, and then update the function to make better prediction on future similar Mondays. With Sutton's method, the weatherman would make a second prediction of Saturday's rain on Tuesday. The temporal-difference error between Monday and Tuesday's predictions could be used to improve predictions for similar Mondays. Sutton refers to the intuition of

temporal-difference learning as "learning a guess from a guess". TD methods are incremental, so they require fewer computational resources than their counterparts.

With other approaches, extra work is required to keep track of all the predictions and then to finally update them when their target values are available. Sutton also claims that TD methods are more data efficient than the competing approaches; they converge faster and learn better predictions with limited data (Sutton, 1988).

TD learning is a method that combines Monte Carlo ideas and dynamic programming (DP) ideas (Sutton and Barto, 1998), and takes the best from these two methods: it can learn directly from raw experience and without a model of the environment's dynamics, like Monte Carlo methods and at the same time it can update estimates based in part on other learned estimates, without waiting for a final outcome like DP do. These methods can also be combined like in $TD(\lambda)$ algorithm, which integrates TD and Monte Carlo methods. It is important to highlight that the differences in these methods are based in the way they approach the prediction problem (policy-evaluation) consists on estimating the value function V^{π} for a given policy π. Obviously, TD methods have an advantage over DP methods in that they do not require a model of the environment, of its reward and next-state probability distributions.

Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy π, both methods update their estimate V of V^{π}. If a non-terminal state s_t is visited at time t, then both methods update their estimate $V(s_t)$ based on what happens after that visit. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(s_t)$. A simple Monte Carlo method suitable for non-stationary environments can be expressed as

$$V(s_{t+1}) \leftarrow V(s_t) + \alpha[r_t - V(s_t)] \tag{3.5}$$

where r_t is the actual return following time t and α is a constant step-size parameter. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(s_t)$ (only then is r_t known), TD methods need wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward r_{t+1} and the estimate $V(S_{t+1})$. In order to express the simplest TD method, known as $TD(0)$, let r_t be the reinforcement on time step t and R_t the correct prediction that is equal to the discounted sum of all future reinforcement. The discounting is done by powers of factor of γ such that reinforcement at distant time step is less important.

$$R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \tag{3.6}$$

where $0 \le \gamma < 1$.

The signal x_t is the information available to the system at time t to enable it to make the prediction p_t. In other words, p_t is a function of x_t, and we can write $p_t = P(x_t)$, where $P(x_t)$ is a prediction function and P_t denote the prediction function at step t, the goal of the algorithm is to update P_t to a new prediction function P_{t+1} at each step. The discount factor determines how strongly future values of r influence current predictions. The problem, then, is to select a function P so that $p_t = P(x_t) \approx R_t$ as closely as possible for $t = 0, 1, 2, \ldots$. This is called, the infinite-horizon discounted prediction problem.

$$R_t = r_{t+1} + \sum_{i=1}^{\infty} \gamma^i r_{t+i+1} \tag{3.7}$$

by changing the index of i to start from 0.

$$R_t = r_{t+1} + \sum_{i=0}^{\infty} \gamma^{i+1} r_{t+i+2} \tag{3.8}$$

$$R_t = r_{t+1} + \gamma \sum_{i=0}^{\infty} \gamma^{i} r_{t+i+2} \tag{3.9}$$

$$R_t = r_{t+1} + \gamma R_{t+1} \tag{3.10}$$

Thus, the reinforcement is the difference between the ideal prediction and the current prediction.

For $t = 0,1,2,\ldots$. Therefore, since $P_t(x_{t+1})$ is the estimate of R_{t+1} available at times-step t, one can estimate R_t by the quantity

$$R_t = r_{t+1} + \gamma P_t(x_{t+1}) \tag{3.11}$$

That is, the current prediction function, P_t, applied to the next input, x_{t+1}, multiplied by γ, gives an estimate of the part of the prediction target that would otherwise require waiting forever to obtain.

It is possible then, to express the temporal difference error δ_{t+1} as:

$$\delta_{t+1} = r_{t+1} + \gamma P_t(x_{t+1}) - P_t(x_t) \tag{3.12}$$

In effect, the target for the Monte Carlo update is r_t, whereas the target for the TD update is $r_{t+1} + \gamma V_t(s_{t+1})$. The most obvious advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an online, incremental fashion. With Monte Carlo methods, one must wait until the end of an episode, because only then the return will be known, whereas with TD methods one need

only wait one time step. Surprisingly often, this turns out to be a critical consideration. Some applications have very long episodes, so that delaying all learning until an episode's end is just too slow. Other applications are continual and have no episodes at all. Finally, some Monte Carlo methods must ignore or discount episodes on which experimental actions are taken, which can greatly slow learning. TD methods are much less susceptible to these problems because they learn from each transition regardless of what subsequent actions are taken.

TD methods learn their estimates in part on the basis of other estimates. They learn a guess from a guess, without waiting for an actual outcome (they bootstrap). While doing this is certainly convenient, the real issue is if convergence to the correct answer can still be guarantee.

For any fixed policy π, the TD algorithm described above has been proven to converge to V^{π}, in the mean for a constant step-size parameter if it is sufficiently small, and with probability one if the step-size parameter decreases according to the usual stochastic approximation conditions (Barricelli, 1954).

If both TD and MC methods converge asymptotically to the correct predictions, then the obvious next issue becomes which method learns fastest. At the current time this is an open question in the sense that no one has been able to prove mathematically that one method converges faster than the other (Sutton and Barto, 1998). In practice, however, TD methods have consistently been found to converge faster than MC methods on stochastic tasks.

3.8.1.1 Q-Learning

One of the most important off-policy TD control algorithms is Q-Learning, originally developed by (Watkins, 1989), is a form of model-free reinforcement learning that can also be viewed as a method of asynchronous dynamic programming (DP). In its simplest form, 1-step Q-Learning, it is defined by:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right] \quad (3.13)$$

Q-Learning provides agents with the capability of learning to act optimally in Markovian domains by experiencing the consequences of actions, without requiring them to build maps of the domains. It works by learning an action-value function that gives the expected utility of taking a given action in a given state and following a fixed policy thereafter. One of the strengths of Q-learning is that it is able to compare the expected utility of the available actions without requiring a model of the environment.

In Q-learning, selecting the action with the best Q-value is necessary only at the update step, which classifies it as an off-policy method. The behaviour is separated from the learning process so any exploratory action can be taken while learning is progressing. After the Q-values have converged, the action with the best Q-value is always selected as in all reinforcement learning methods. Therefore, it can be said that Q-learning learns the greedy policy (the policy expected to result in the optimal policy).

A pseudo-code example of a Q-Learning algorithm is presented in Table 3.2, (Sutton and Barto, 1998).

Additionally, considering a problem with discrete states and actions where the Q-function can be represented as a table with one entry for every state-action pair, it has been shown that under certain boundary conditions Q converges with probability 1 to the optimal value function, Q* (Watkins, 1989, Watkins and Dayan, 1992). These boundary conditions are:

- At time step t, the learning rate α_t, must have the following characteristics:

$$\sum \alpha_t = \infty$$
$$\sum \alpha_t^2 < \infty \tag{3.14}$$

- The environment must be finite, Markovian, and stationary.

Under this conditions, if the agent visits all the states and picks all the actions an infinite number of times and the policy converges in the limit to the greedy policy (3.13) the Q estimate will converge to the optimal, true Q-function Q* with probability 1.

TABLE 3.2 Q-LEARNING ON-POLICY TD CONTROL ALGORITHM

Algorithm

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
  Choose a from s using policy derived from Q
  Take action a, observe r, s'
  Choose a' from s' using policy derived from Q
  $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a_t) - Q(s_t,a_t)\right]$
  $s \leftarrow s'$; $a \leftarrow a'$
  Until s is terminal

Other convergence studies include:

- Assume unique maxima of Q (Littman, 2001)

3.8.1.2 SARSA

The SARSA (State-Action-Reward-State-Action) algorithm (Rummery and Niranjan, 1994) is a temporal difference (TD) method for learning a Markov decision process policy (e.g. learns action-value functions by making estimations based on previous estimations rather than learning a state-value function) and is almost identical to Q-learning. SARSA was created as an alternative to Q-learning

(Watkins and Dayan, 1992), which, as seen in the previous section, it updates the policy based on the maximum reward of available actions. The difference between these two methods is that SARSA learns the Q values associated with taking the policy it follows itself, while Q-learning learns the Q values associated with taking the exploitation policy while following an exploration/exploitation policy.

For an on-policy method we must estimate $Q^{\pi}(s,a)$ for the current behaviour policy π for all states s and actions a according to:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right] \qquad (3.15)$$

where s_t is the state of the agent at time t, a_t is the action chose by the agent, r_{t+1} is the reward the agent gets for choosing action a_t, s_{t+1} is the state that the agent will now be in after taking that action, and finally the action a_{t+1}, is the action the agent ends up taking. The SARSA learning agent will interact with the environment and update its policy based on the actions taken. The Q value $Q(s_t,a_t)$ represents the possible reward received at time step $t+1$ for taking action a in state s, plus the discounted future reward received from the next observation. In (3.15) this Q value is updated by a temporal difference error, with a learning rate α.

If s_{t+1} is terminal, $Q(S_{t+1},a_{t+1})$ is defined as zero. As in all on-policy methods, we continually estimate Q^{π} for the behaviour policy π, and at the same time change π towards greediness with respect to Q^{π}.

TABLE 3.3 SARSA ON-POLICY TD CONTROL ALGORITHM

Algorithm
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):

Initialize s
Choose a from s using policy derived from Q
Repeat (for each step of episode):
  Take action a, observe r, s'
  Choose a' from s' using policy derived from Q
  $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
  $s \leftarrow s'$; $a \leftarrow a'$
Until s is terminal

A pseudo-code example of a SARSA algorithm is presented in Table 3.3 (Sutton and Barto, 1998).

In 2000 (Singh et al., 2000) demonstrated the convergence of the one step SARSA algorithm to the optimal solution. Considering the same boundary condition described in the previous section, optimality is reached by using $\varepsilon - greedy$ policies, were greedy actions are chosen most of the times with a small probability ε, while some other non-greedy actions are chosen randomly. This is done to achieve the right balance between exploration and exploitation so that after an infinite number of trials (infinite experience), all state-action pairs are visited an infinite number of times so that, in the limit, the policy converges to the greedy policy.

3.9 Reinforcement Learning In Decision Making

Decision making can be considered an important skill for animals and humans acting and surviving in environments only partially understood, where rewards and punishments are assigned for their successes or failures (Dayan and Daw, 2008). Conclusive studies have shown that is a primitive neural network model of the Basal Ganglia system in animals, that slowly learns to make decisions on the basis of the relative probability of rewards (Frank and Claus, 2006) as it have being shown by a series of influential models proposing that the responses of

dopamine neurons can be identified with the error signal from temporal difference (TD) learning (Daw, 2003).

In artificial systems, decision-making environments are characterized by a few key concepts: a set of decisions, a state space, a set of actions, and outcomes. The actions can move the decision-maker from one state to another (i.e. induce state transitions) producing outcomes. These outcomes are assumed to have a numerical (positive or negative) utility, which can change according to the motivational state of the decision-maker or by direct experimental manipulation.

Typically, the decision-maker starts off not knowing the rules of the environment and has to learn or sample these from experience as in a Markov decision process (MDP). In a MDP, a decision maker is faced with the problem of correcting the behaviour of a probabilistic system as it evolves through time. In a MDP the agent can choose an action with the goal of computing a sequence of actions that will result in the system performing optimally with respect to some predetermined performance criterion (Gardiol and Kaelbling, 2006).

Decision making algorithms are very useful tools for many fields, ranging from finances to control, and reinforcement learning algorithms have proven to be a very useful tool for finding value functions for future decisions. Decision theory has been combined with reinforcement learning algorithms in order to manage uncertainty in state-transitions as in (Gaskett, 2003). This approach was further improved in (Pednault et al., 2002, Rogova et al., 2002) where the use of cost-sensitive sequential algorithms improved the performance when there is uncertainty regarding the selection of future actions.

3.10 Reinforcement Learning For Control

An important advantage of learning in control systems is that learning can reduce or completely eliminate the need for an accurate model of the dynamics of a system or its environment. As it has been seen before, reinforcement learning does not require the model of the system or the environment to perform, rather is solely guided by a reinforcement signal. When it is applied to control, the policies are fine-tuned based on performance errors (An et al., 1988) as it will be explored in the DC motor example in Chapter 6.4. The problem of control is the problem of finding the optimal policy π^*, which can be accomplished by finding the optimal action value function Q^*.

Reinforcement learning (RL) has been shown to be an effective technique for deriving controllers for a variety of systems, such as:

- Cart-pole balancing problem (Barto et al., 1983), where the goal is to stabilize a vertical pole by applying an horizontal force to a car. The reinforcement learning task in this experiment consists on finding the direction of this force (to the left or right) for every system state, derived from the cart position, speed and the pole angle and radial speed.

- Mobile robots (Willgoss and Iqbal, 1999, Gaskett et al., 2000). In this study a reinforcement learning algorithm is implemented in small mobile robots, in order to learn behaviours to that can allow the robot to complete a certain assigned task under noisy infrared information.

- Motor control (Hinojosa et al., 2008). In this study, a fuzzy-reinforcement learning algorithm is applied to control a DC motor position by learning the rule matrix of a fuzzy controller.

3.11 Conclusions

Reinforcement learning provides a set of very useful methods for learning in cases there the model of the environment or system is not available. The most remarkable advantage in the use of reinforcement learning methods is in its use on systems where complete feedback information is not available, as RL only requires information about the success or failure of an action.

Temporal difference learning is a prediction method, mostly used for solving the reinforcement learning problem. TD learns how to predict a value that depends on future values of a given signal. It was also seen that the SARSA and Q-learning algorithms are very similar, while SARSA updates $Q(s,a)$ for the policy it is actually executing, Q-learning updates $Q(s,a)$ for a greedy policy (uses max(a) to pick action to update). It can be said that Q-learning will learn the "true" optimal policy, but SARSA will learn about what it's actually doing.

The two main fields of application of reinforcement learning are decision making and control problems. So far reinforcement learning have been used successfully in several different applications under both fields, yet some more practical and real-world applications need to be tested.

3.12 Summary

This chapter explained the reinforcement learning process in the human brain. Then, it introduced the principles of reinforcement learning and important

common terms used in the reinforcement learning literature. Additionally a complete classification of several reinforcement learning methods was provided. Two important temporal difference-based algorithms for reinforcement learning have been described: Q-learning and SARSA. This chapter concludes explaining the use of reinforcement learning in decision making and control systems.

The next chapter will introduce the concept of uncertainty and explains several methods developed for dealing with this issue. It also provides an introduction to fuzzy logic and probabilistic theory as two important methods for dealing with uncertainty.

Chapter 4

Probabilistic Fuzzy Inference Systems

4.1 Introduction

As we have seen in the previous chapter, reinforcement learning can be a powerful tool in the decision making process when the learning agent is dealing with stationary observable environments. But in reality, stationary and observable environments are only possible in virtual worlds (simulation). When the learning agent is embodied and interacting with a real environment uncertainty is unavoidable. The inputs are prone to electrical noise, systematic errors, and different kinds of perturbations that will make them inaccurate. At the same time, the outcome of the actions taken is not always fully predictable. This unpredictability becomes evident for cases where the agent has to deal with uncharted or unknown stochastic environments.

In the last few decades, fuzzy logic have been the focus of interest of researchers as a way to deal with noisy input data in a natural human-like way, proving to be

an excellent tool to deal with non-statistical uncertainty. It became obvious that the combination of probabilistic theory with fuzzy logic would be the next step up, covering the uncertainty gap, by providing a way of handling statistical uncertainty. Probabilistic fuzzy logic systems have quickly become the focus of modern research with applications in areas such as finance, weather forecast and robotics.

The rest of this chapter is organized as follows; section two presents an introduction to uncertainty defining its common sources and common methods to deal with this issue. Then, in section three, an overview of fuzzy logic systems including some common learning methods used to optimize fuzzy inference systems is presented. Following, in section four, an introduction to probabilistic theory is given and in section five, the advantages of combining probabilistic theory and fuzzy logic into probabilistic fuzzy logic is explained. Section 6 presents some important learning methods for fuzzy logic and outlines an introduction of reinforcement learning applied to fuzzy logic. Sections seven and eight provide the relevant conclusions and summary.

4.2 Uncertainty

We humans, are constantly deciding many of our actions without having complete data about the facts we are analysing, in other words, we must take decisions in the light of uncertain knowledge about a situation. As a consequence questions like how do I fill the gaps in my own knowledge?, what should I do when not all the details are known? And, how should I deal with unpredictable events? These are questions every human must constantly face. In the same way this are also issues that learning agents have to be able to handle. This set of questions constitutes what it will be referred since now on as uncertainty. But, what exactly is uncertainty?

Uncertainty is a term used to identify situations or states where the perceived environment or the outcome of an action cannot be precisely described. It is originated from insufficient information (vagueness) or from redundant discordant information (ambiguity) about the world and can be produced by many reasons, including: inherent limitations to model the world, noise, perceptual limitations in a physical agent, like error in sensor measurements and the approximate nature of many algorithmic solutions embedded in this agents.

Uncertainty comes in many different ways and in the real world, it is impossible to avoid uncertainties. When designing intelligent systems, the problem of how to handle uncertain, imprecise and incomplete information arises. According to (Dwivedi et al., 2006) uncertainty arises at different levels. At an empirical level, uncertainties are associated with the errors in measurements and resolution limits of the sensors. At a cognitive level, it arises from the vagueness and ambiguities in natural language. At a social level, uncertainty is created and maintained by people for different reasons, like privacy.

4.2.1 Uncertainty Taxonomy

In 2000, (Regan et al., 2000) proposed that uncertainty can be group in two categories: epistemic uncertainty and linguistic uncertainty, where epistemic refers to uncertainties originated by indeterminate facts and linguistic uncertainty refers to uncertainties originated by indeterminacy in the language. Later in 2007, (Tannert et al., 2007) proposed a new categorization that considered other sources of uncertainties as it is depicted in Figure 4.1. In this same figure, the dark shaded boxes indicate the classification of the uncertainties that this work addresses.

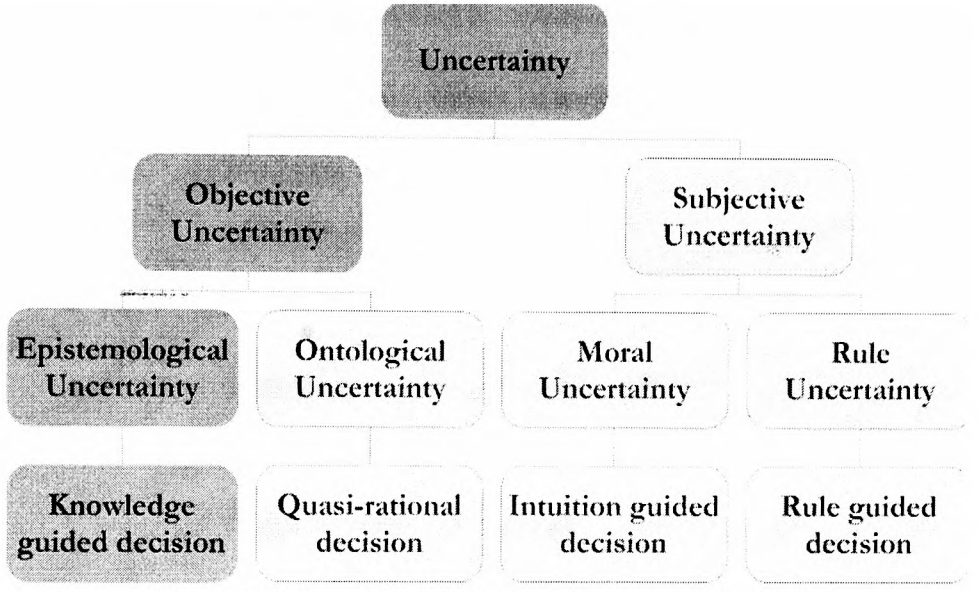

Figure 4.1: Uncertainty taxonomy (Tannert et al., 2007).

4.2.2 Sources of Uncertainty

Uncertainty arises from many different sources, as a result, the method used to handle uncertain information in a system have direct impact on its performance. In the case of the Epistemological uncertainty, the sources of uncertainty can be grouped in four categories:

- Noise and Conflicting Data:
    - Observations of the environment are not precise. Sensors in physical agents induce randomness on its output.
    - Measurement errors.
    - Systematic errors
- Uncertain change:

- The world does not behave deterministically.
- Successor states cannot be predicted.
- Physically embodied agents are unpredictable (e.g. wear and tear).

- Incompleteness ("ignorance"):
  - Incomplete domain knowledge.
  - Incomplete information (e.g. models of physical things are only approximate).

- Selection among the alternatives.

As a contrast with epistemological uncertainty, linguistic uncertainty as described by (Regan et al., 2000) can be originated due to:

- Vagueness, constitutes a form of uncertainty and it is defined as the character of which with contours or limits lacking precision or clearness (Bacon, 2009). Bertrand Russell was the first to have discussed of them in the bald man paradox; at the beginning, man is not bald. Then he loses his hair one by one; at the end, he is a bald person. Therefore it should be a hair whose loss converted the man into a bald person. This is of course absurd. In truth the concept of bald person is vague. There are men certainly bald, others not, and between them, there are men for whom this is not completely true to define them like bald or not (Sainsbury, 1995).

- Ambiguity is the uncertainty due to lack of clarity and/or lack of precision. Is a term used in under conditions where information can be understood or interpreted in more than one way and is distinct from vagueness, which is a statement about the lack of precision contained or available in the information. Can be due to purposive strategy (e.g., to confuse) or by lack of knowledge. There are two types of ambiguity :
    - Discord (conflict) is the disagreement in choosing among several alternatives, and
    - Non-specificity is the uncertainty related to alternatives represented by a set. It is present when two or more alternatives are left unspecified.
- Context dependency. In phonetics the dependency on the context about what it is being said can generate uncertainty on the information that it carries.

4.2.3 Dealing with Uncertainty

In science and technology, the ultimate goal of a learning agent (e.g. a physical agent) is to obtain true information (e.g. an accurate representation of the surrounding world), so that it can to perform effectively by taking an appropriate action. However, this is impossible due to imperfections in measurements, missing information or model mismatches. If this uncertainty could be significantly reduced, we will become more "confident" that a desirable event will happen, or that an undesirable event will not. If an accurate representation of this world is not attainable, then an effective way to deal with this inaccuracy is needed.

To date there are several methods for dealing with uncertainty. According to (Akerkar, 2005) these can be grouped in three categories:

- Fuzzy Logic Methods
- Symbolic Methods
    - Non-monotonic reasoning
    - Possibility theory
- Statistical Methods
    - Certainty factors
    - Probabilistic theory
    - Dempster-Shafer theory

4.2.3.1 Fuzzy Logic Methods

One of the first attempts to represent uncertainty was introduce with the development of fuzzy set theory (Zadeh, 1965). Fuzzy logic theory focuses on ambiguities in describing events rather than uncertainties about the occurrence of an event (Akerkar, 2005), due to this, it fails to handle uncertainty by itself, as it only handles degrees of truth. Later on, (Zadeh, 1978) developed a broader framework for uncertainty representation called possibility theory (a symbolic method), which is also known as a fuzzy measure. In (Zadeh, 1978) a normal fuzzy membership function was interpreted as a possibility measure.

4.2.3.2 Symbolic Methods

Symbolic methods represent uncertainty belief as being: true, false or neither true or false. Symbolic methods have the advantage of using a well-defined reasoning

and inference framework with a simple mechanism, but it does not provide a quantitative notion of uncertainty; whilst statistical methods try to represent beliefs that are uncertain but for which there may be some supporting or contradictory evidence (Akerkar, 2005) and offers advantages in genuine randomness and exceptions scenarios. We can contrast statistical with symbolic methods observing that probability theory is based on one type of measure: the probability measure. Whilst possibility theory is based on two types of measures: the possibility and the necessity measure (Dubois, 2006).

4.2.3.3 Statistical Methods

Statistical methods provide a method for representing beliefs that are not certain (or uncertain) but for which there may be some supporting (or contradictory) evidence. Statistical methods can be used to summarize or describe a collection of data; in addition, patterns in this data may be modelled in a way that accounts for randomness and uncertainty in the observations, and then, used to draw inferences about the process being studied.

- **Certainty factors** are mostly used in areas where expert's judgments a re a vailable, whil st pr obability the ory is use d in cases where statistical data is available. A certainty factor is a number $\in[-1,1]$ that reflects the degree of belief in a hypothesis, positive certainty factors indicate that there is evidence that the hypothesis is valid. The larger the certainty factor, the greater is the belief in the hypothesis. When CF=1, the hypothesis is known to be correct. A CF=-1 implies that the hypothesis is effectively disproved, CF=0 implies that there is neither evidence regarding the hypothesis being true or false.

- **Probability theory** is the oldest and possibly still best developed. Several other theories are based on or are generalizations of probability theory, such as upper and lower probabilities, certainty factors or Bayesian networks.

- **Dempster-Shafer Theory (DST)** is a mathematical theory of evidence. The seminal work on the subject is done by Shafer (Shafer, 1976), which is an expansion of previous work done by Dempster (Demspster, 1969). The theory came to the attention of AI researchers in the early 1980s, when they were trying to adapt probability theory to expert systems. The DST is based on two ideas: the idea of obtaining degrees of belief for one question from subjective probabilities for a related question, and Dempster's rule for combining such degrees of belief when they are based on independent items of evidence. In a finite discrete space, Dempster-Shafer theory can be interpreted as a generalization of probability theory where probabilities are assigned to sets as opposed to mutually exclusive singletons. In traditional probability theory, evidence is associated with only one possible event, whilst in DST, evidence can be associated with multiple possible events.

4.3 Fuzzy Logic

The idea of fuzzy logic (FL), was originally introduced by Prof. Lotfi Zadeh and it was presented not as a control methodology, but rather as a way of processing data by allowing partial set membership rather than crisp set membership or non-membership (Zadeh, 1973).

FL, similarly to some other scientific theories, is not a new theory but rather, an extension to previous theories, in particular, to the conventional Boolean logic. As a contrast with traditional crisp logic, FL is based on fuzziness and uncertainty and concerns with the general concept of "degree of truth" in the sense that the degree of truth is no longer limited to zero and one. FL's core concept was founded atop the fuzzy set theory (Zadeh, 1965), in which belonging to a set is not a binary rough criteria, but rather it has some uncertainty inside that allows a partial membership.

This particular characterization of fuzzy logic systems provides them with several advantages. Firstly, fuzzy logic systems have the ability to represent and manipulate linguistic variables and sentences with a natural language. This feature enables us to incorporate human expert knowledge in the form of fuzzy if-then rules and fuzzy membership functions. Secondly, fuzzy logic systems have an outstanding capability to map the typical nonlinear relation of input-output model without a precise mathematical formulation (Liu and Li, 2005).

Besides these two remarkable characteristics, fuzzy logic systems are able of handling and representing non-statistical uncertainty (Meghdadi and Akbarzadeh-T., 2001). The universal approximation property of fuzzy logic systems guarantees their ability for modelling deterministic complex and uncertain systems. These superior traits however can be degraded by the existences of statistical (e.g. randomness) and probabilistic uncertainties.

Other key characteristics of FL systems are:

- Based on natural language (imprecise but very descriptive)
- Easy to understand.
- Based on very simple mathematical concepts.

- Flexibility.
- With any given system, it is easy to add on more functionality without starting again from scratch.
- Very robust and forgiving of operator and data input errors.
- Ability to model nonlinear functions of arbitrary complexity.
- Present a convenient way to map from an input space to an output space.
- It can be built on top of human expertise (capable of formulating expert's knowledge)
- Easy interpretation of the results, because of the natural rules representation.
- Easy extension of the base of knowledge through the addition of new rules.
- Can be blended with conventional control techniques, not necessarily to replace them, but in many cases, to augment and simplify their implementation.

And the most important drawbacks of FIS are:

- Incapable to generalize, as it only answers according to what is written in its rule base.
- Not robust against topological changes of the system (stochastic environments), such changes would demand alterations in the rule base.

- Depends on the existence of an expert to determine the inference logical rules.

4.3.1 Fuzzy Inference Systems

A FIS is the process of formulating a mapping from a given input to an output using fuzzy logic, as it can be seen in Figure 4.2. This mapping then provides the basis from which decisions can be made, or patterns classified (McNeill and Thro, 1994). Therefore, it is possible to classify FIS as expert systems with actions based on a set of rules, were the antecedents and the consequents are expressed in linguistic terms, resembling human natural language. This important characteristic makes FIS very easy to understand and allows the operator to incorporate human (*"a priori"*) expert knowledge in the form of fuzzy if-then rules and fuzzy membership functions.

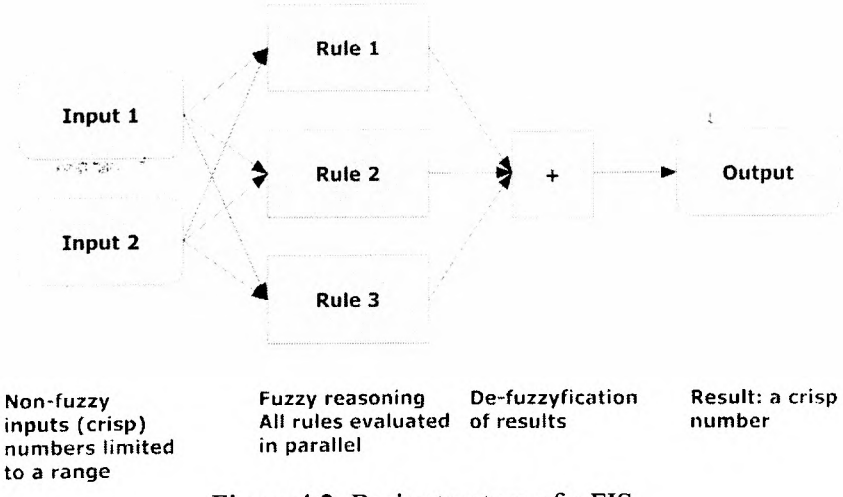

Figure 4.2: Basic structure of a FIS.

FIS has been successfully applied in many different fields ranging from automatic control and expert systems to data classification, decision analysis and computer vision. Due to this multidisciplinary nature, FIS became associated with a rather

large number of methods such as fuzzy-rule-based systems, fuzzy expert systems (Siler and Buckley, 2004), fuzzy modelling, fuzzy associative memory (Kosko, 1991), and the one of especial concern for this work, fuzzy logic controllers.

When a FIS is used for controlling systems it is called fuzzy logic controller (FLC). FLC are especially advisable for cases where the mathematical model of the system to be controlled is unavailable, and the system is known to be significantly nonlinear, time varying, or to have a time delay. Within the area of automatic control, FIS have had great success in the field of robotics. FLC are particularly suitable for implementing systems with stimulus-response behaviour, since fuzzy rules provide a natural framework to describe the way the system "should react" whilst providing human reasoning capabilities in order to capture uncertainties (Jang et al., 1997).

One of the first control systems built using fuzzy set theory (and one of the most commonly used) is the Mamdani's fuzzy inference method (Mamdani and Assilian, 1975). Mamdani's effort was based on Zadeh's paper on fuzzy algorithms (Zadeh, 1973) for complex systems and decision processes.

Another important method is the Sugeno inference method. In general, Sugeno systems can be used to model any inference system in which the output membership functions are either linear or constant.

4.4 Probabilistic Theory

From all the methods for dealing with uncertainty mentioned earlier in this chapter, probabilistic theory is the oldest (can be back traced to the early 1960s) and the best understood of all. As a contrast with FL, probability theory concerns with the concept of "probability of truth" and gives information about the likelihood of an event in the future by representing information through

probability densities. Therefore, probability and fuzziness are concepts that represent two different kinds of uncertainty, statistical and non-statistical, respectively.

In recent years this probabilistic approach has become the dominant paradigm in a wide array of problems, ranging from financial (Berg et al., 2004, Almeida and Kaymak, 2009), control (Liu and Li, 2005, Blackmore, 2006, Hinojosa et al., 2008), robotics (Thrun, 2000, Valavanis and Saridis, 1991, Park et al., 2007, Thrun et al., 2000, Jaulmes et al., 2005), and for representing uncertainty in mathematical models (Ross, 2004). Some research work, as in (Cheeseman, 1985), supports the idea that all the numerous schemes for representing and reasoning about uncertainty featured in the AI literature are unnecessary as probability theory can easily and effectively deal with this issue.

Probability theory attempts to quantify the notion of probable. The general idea is divided into two concepts:

- **Aleatory (objective) probability**, which represents the likelihood of future events whose occurrence, is governed by some random phenomena.

- **Epistemic (subjective) probability**, which expresses the uncertainty about the outcome of some event, in the lack of knowledge or causes.

If we apply the concepts of probabilistic theory to the particular case of an embodied agent, this approach becomes divided in two fields, (Thrun, 2000):

- **Probabilistic perception:** Deals with the uncertainty about the external world, as captured by the sensors of the embodied agent

by using a probability distribution of the captured data instead of discreet values.

- **Probabilistic control:** Given the uncertainty about the environment, a learning agent faces the task of taking a decision about its next action which consequences might be uncertain, especially over the long-term.

In this same particular case, a probabilistic controller will need to anticipate to various contingencies that might arise in uncertain worlds, by blending information gathering (exploration) with robust performance-oriented control (exploitation). In this case, the uncertainty of the state is propagated forward in order to obtain a probabilistic representation of a long-term behaviour.

There are two broad views on probability theory for representing uncertainty: the frequentist and the subjective or Bayesian view.

- **The Frequentist View,** sometimes also referred as "empirical" or "*a posteriori*" view of probability, relates to the situation where an experiment can be repeated indefinitely under identical conditions, but the observed outcome is random. Empirical evidence suggests that the relative occurrence of any particular event, i.e. its relative frequency, converges to a limit as the number of repetitions of the experiment increases. Therefore, probabilities are defined in the limit of an infinite number of trials.

- **The Subjective View**, was originally introduced by (Pearl, 1982, Pearl, 1988) and further developed later by (Lauritzen and Spiegelhalter, 1988). The subjective or Bayesian view of probability is used as a belief where the basic idea in the

> application of this approach is to assign a probability to any event based on the current state of knowledge and to update it in the light of the new information. The conventional procedure for updating a prior probability in the light of new information is by using Bayesian theorem where subjective probabilities quantify degrees of belief.

Other less common views can include the classical view, sometimes referred as "*a priori*" or "theoretical" and the axiomatic view, which is a unifying perspective aimed to provide a satisfactory formal structure for the development of a rigid theory by focusing on the question "How does probability work?" rather than trying to define what probability is.

As probabilistic methods and fuzzy techniques are good for processing uncertainties (Zadeh, 1995, Laviolette and Seaman, 1994), it would be beneficial to endow FLS with probabilistic features. The integration of probability theory and fuzzy logic has been the subject of many studies as in (Liang and Song, 1996). In the next section probabilistic fuzzy logic systems will be described in more detail.

4.5 Probabilistic Fuzzy Logic Systems

In the previous chapter we have seen that statistical uncertainty may be viewed as a kind of uncertainty concerning the occurrence of an event in the future. Statistical uncertainty is best represented with probability, which gives us the likelihood of occurrence of an outcome, presented in a statistical manner. However, there are other types of uncertainty present, for example, the uncertainty related to perception. These other types of uncertainty should best be modelled

explicitly by using paradigms other than probabilistic modelling (Berg et al., 2004).

On the other hand, fuzzy systems have two important traits: a universal approximation property, which guarantees the ability for modelling deterministic, complex, and uncertain systems (non-stochastic uncertainty); and its inherent ability for handling and representing non-statistical uncertainties. These superior traits however can be severely degraded by the existence of statistical uncertainties (like randomness) and probabilistic elements (Meghdadi and Akbarzadeh-T., 2001).

In order to deal with these issues, the concept of probabilistic fuzzy logic (PFL) emerges with the goal of enhancing the universal applicability of fuzzy systems by bridging the gap between fuzziness and probability. The need for this integrated framework is highlighted in situations where both types of uncertainty exist concurrently and where each of the fuzziness and probability concepts alone are necessary but not sufficient, e.g. the representation of non-deterministic real world systems.

Probabilistic fuzzy logic systems work in a similar way as regular fuzzy logic systems and encompass all their parts: fuzzification, aggregation, inference and defuzzification but they incorporate probabilistic theory to improve the stochastic modelling capability. Already in the early 80's, Zadeh defined probabilities on discrete fuzzy sets, (Zadeh, 1984). The same idea is continued by (Pan et al., 1996), where Bayesian inference is applied to fuzzy distributions.

In order to model uncertainty, (Meghdadi and Akbarzadeh-T., 2003) defined probabilities on the rules of fuzzy systems. In this case, a fuzzy rule will be expressed as follows:

$$R_j: \textit{If } x_1 \textit{ is } A_1^h \ldots x_i \textit{ is } A_i^h \ldots \textit{and } x_I \textit{ is } A_I^h \textit{ then "}y\textit{" could be } B_1 \textit{ with a probability of success of } \rho_{j1}, \ldots, B_k \textit{ with a probability of success of } \rho_{jk} \textit{ and } B_n \textit{ with a probability of success of } \rho_{jn}$$

where R_j is the j^{th} rule of the rule base, A_i^h is the h^{th} linguistic value for input i, and $h=\{1,2,\ldots,q_i\}$ where q_i is the total number of membership functions for input x_i. Variable y denotes the output of the system and B_k is a possible value for y, with $k=\{1,2,\ldots,n\}$ being the action number and n is the total number of possible actions that can be executed. The probability of this action to be successful is ρ_{jk}, where $j=\{1,2,\ldots,m\}$ is the rule number and m is the total number of rules. So that $\rho_{j1}+\rho_{j2}+\cdots+\rho_{jn}=1$.

According to (Berg et al., 2002) a method for defining probabilities on fuzzy sets is developed by first defining the probability of a singleton fuzzy event as:

$$P(S_i)=\mu_{S_i} f(x_i)=\sum_{x_k \in X} \mu_{S_i}(x_k) f(x_k) \tag{4.1}$$

In (4.1), $f(x_k)=P(x_k)$ by definition and x_k is a fuzzy sample. If the above definition is extended to a countable set of discrete fuzzy events A_b defined on the sample space Ω a vector of membership values is obtained, where:

$$\mu_{A_b}(x_k)=m_{A_b,k} \tag{4.2}$$

Then for a fuzzy partition it can be said that:

$$\sum_{A_b} \mu_{A_b}(x_k)=1 \quad \forall x_k \tag{4.3}$$

Now we can rewrite several probability theory properties for fuzzy sets.

- The membership function of the intersection:

$$\mu_{A_b \cap B_c}(x_k) = \mu_{A_b}(x_k)\mu_{B_c}(x_k) \tag{4.4}$$

$$P(A_b \cap B_c) = \sum_{x_k \in X} \mu_{A_b}(x_k)\mu_{B_c}(x_k) f(x_k) \tag{4.5}$$

The conditional probability is shown in (4.6), where x_p is a finite set of representative samples.

$$P(A_h \mid B_c) = \frac{\sum_{x_p} \mu_{A_b}(x_p)\mu_{B_c}(x_p)}{\sum_{x_p} \mu_{B_c}(x_p)} \tag{4.6}$$

And finally we can express the total law of probability as:

$$\sum_{A_h} P(A_h \mid B_c) = 1 \tag{4.7}$$

(Liu and Li, 2005) who applied it to solve a function approximation problem and a control robotic system, showing a better performance than an ordinary FLS under stochastic circumstances. Other PFS applications include classification problems (Berg et al., 2002) and financial markets analysis (Berg et al., 2004).

This new concept not onl y has the advantages of the approximate reasoning property of fuzzy systems, but it can also be regarded as an extension to conventional FL in the sense that the latter is a special case of the former with zero degree of randomness. In a PFL the truth values are not only specified with a degree of truth between zero and one, but also with a probability of truth in the form of a probability number or a probability distribution. Consequently, in a PFL system, the degree of truth and the probability of truth are simultaneously

considered. Therefore, it is expected from this PFL to have an increased capability in handling both, statistical and non-statistical uncertainty, simultaneously.

4.6 Learning Methods for Fuzzy Systems

Fuzzy logic is an effective tool for solving many nonlinear control problems, where the nonlinear behaviour of the system makes it difficult, if not impossible, to build an analytical model of the system. Additionally, fuzzy set theory provides a mathematical framework for modelling vagueness and imprecision. However, building a fuzzy controller has its own difficulties.

The process of designing a FLC has two challenging tasks: defining the controller structure and, second, finding the numerical values for the controller parameters. These challenges arise due to a lack of a well-established theoretical approach; rather, they are entirely based on the empirical experience of a human operator, which is transferred into the FLC. However, the extraction of the expert's knowledge is not always an easy task; decision rules with complex structures and an excessively large number of variables necessary to solve the control task introduce difficulty in performing the knowledge extraction.

A direct solution to these problems is to use learning algorithms in order to replace or enhance the human operator *"a priori"* knowledge. Fuzzy logic learning can be used to automatically provide a solution for these issues, thereby removing human input from the design.

Several techniques reported in recent literature to create such intelligent controllers include the use of algorithms such as neural networks, genetic algorithms, and more recently reinforcement learning, in order to learn and optimize a fuzzy logic controller.

The result of combining FIS and learning algorithms has several important advantages, such as higher robustness, shorter development time, and less assumptions about the dynamical behaviour of the plant, that makes it attractive for application to real-world problems.

4.6.1 Neuro-Fuzzy Systems

Neural networks (NN) have the ability, to learn complex mappings, generalize information, and classify inputs. The combination of NN with FL results in a hybrid intelligent system that synergizes these two techniques by incorporating the human-like reasoning of fuzzy systems with the learning and connectionist structure of NN (Fuller, 2000). This combination, referred as Neuro-fuzzy hybridization was proposed by (Jang, 1993) and is also known as fuzzy neural network (FNN) or Neuro-fuzzy system (NFS) in the literature. The main strength of Neuro-fuzzy systems is that they are universal approximators with the ability to solicit interpretable IF-THEN rules and involves two contradictory requirements in fuzzy modelling: interpretability and accuracy. This leads to the division of the Neuro-fuzzy research field into two areas: linguistic fuzzy modelling (focused on interpretability), and precise fuzzy modelling (focused on accuracy).

Neural networks and FL can be combined in several different ways depending on the specific requirements and challenges imposed by the control task. According to (Nauck et al., 1997) Neuro-fuzzy systems can be classified in three categories:

- **Cooperative Neuro-Fuzzy System:** The system goes through a pre-processing phase where the NN mechanisms of learning determine some sub-blocks of the fuzzy system like the FIS membership functions or the fuzzy rules. Once the FIS parameters are determined, the NN goes to the background. This is depicted in Figure 4.3 a).

- **Concurrent Neuro-Fuzzy System:** The NN and the fuzzy system work continuously together. In general, the NN assists the FIS continuously to determine the required parameters especially if the input variables of the controller cannot be measured directly. This can be done by pre-processing the inputs and/or post-processing the outputs. This is depicted in Figure 4.3 b).

- **Hybrid Neuro-Fuzzy System:** In the hybrid or fused Neuro-fuzzy system, the NN are used to learn some required parameters of the fuzzy system (parameters of the fuzzy sets, fuzzy rules, and/or weights of the rules), share data structures and knowledge representations. This is depicted in Figure 4.3 c).

The use of NN in combination with FL has provided several benefits and it enjoys real world applicability. However, there are still important drawbacks:

- Requires input-output training pairs, which are not always available.

- Computationally expensive.

- Increased complexity, compared with fuzzy only or NN only methods.

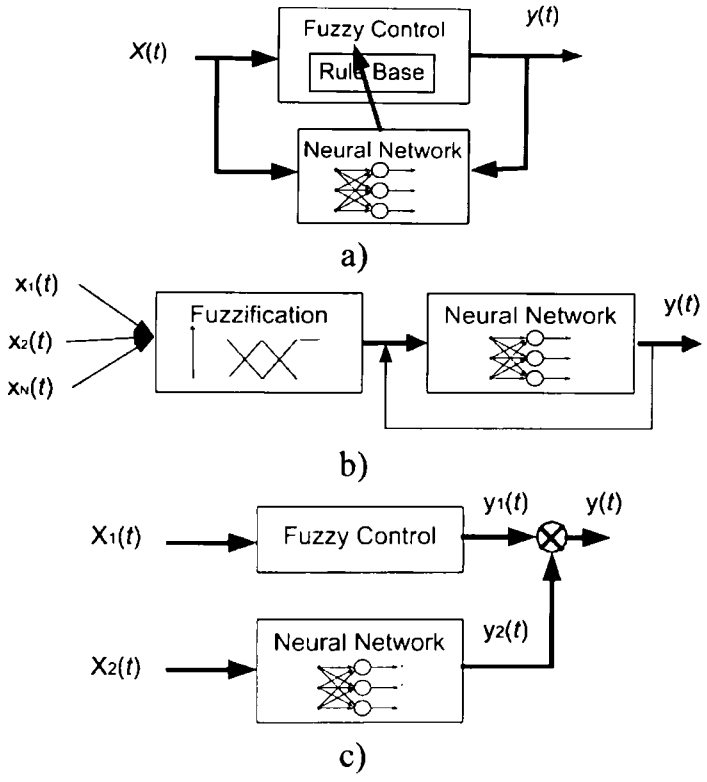

Figure 4.3: Different configurations of neuro-fuzzy systems.

4.6.2 Fuzzy Logic and Genetic Algorithms

As discussed in the previous sections, when designing a FLC, the operator will find two difficulties: how to establish the structure of the controller and, second, how to set numerical values of the controller parameters.

To tackle these problems, a new kind of algorithms called genetic algorithms (GA) was proposed by (Holland, 1992), based on the Darwinian principle of survival of the fittest for reproduction and mutation.

Genetic algorithms (GA) are stochastic search techniques that operate without an explicit knowledge of the task domain by using only the fitness of evaluated

individuals. This feature makes the method universal and allows training of fuzzy systems of arbitrary configuration and of arbitrary sets of fuzzy system parameters (Ng and Li, 1994).

For any given task, the learning process will go through the following steps:

- Take every possible solution and create a population of possible solutions.
- Test different regions of the population in terms of the strength of its solution (i.e. how good it is).
- The strongest solutions in a population will be mated resulting in the creation of offspring, which would replace the weakest solutions of the population.
- Repeat for several generations until the GA cannot find a better solution.

When combined with FLC, there are several ways GA can enhance the FIS overcoming its limitations:

- **Tuning the controller parameters.** According to (Driankov et al., 1996), some of the parameters that can be altered to modify the controller performance are:
    - The scaling factors for each variable.
    - The fuzzy sets representing the meaning of linguistic values.
    - The if-then rules.

- **Learning the controller structure.** The GA is configured to modify the topology of the FLC which can include changes like:
    - Adding/deleting membership functions.
    - Changing the shape of membership functions.
    - Consequently, adding/deleting rules.
    - Changing the type of fuzzyfication/defuzzyfication methods.

The use of GA is a strong alternative for many optimization problems not only for its efficiency, but also for their inherent simplicity and versatility, they are often not difficult to implement onto existing applications. GA are also very time efficient, capable of finding solutions to problems in a short amount of time. According to (Goldberg, 1994) some important advantages include:

- Ability to solve hard problems quickly and reliably.
- Easy to interface to existing simulations and models.
- Extensibility.
- Easy to hybridize.

Although GA provide a good and efficient solution for many optimization problems, its use is still restricted due to some intrinsic limitations such as:

- Certain optimisation problems cannot be solved by means of genetic algorithms. This occurs due to poorly known fitness functions.

- There is no absolute assurance that a genetic algorithm will find a global optimum nor it can assure a constant optimisation response time. Evolution is inductive; in nature life does not evolve towa rds a good solut ion it evolves away from bad circumstances. This can cause a species to evolve into an evolutionary dead end.

- Genetic algorithms have optimisation response times are much larger than other learning methods.

For online control of real systems, random solutions and convergence issues, as the ones described, seriously limit the applicability of GA use in real time applications.

4.6.3 Fuzzy-Reinforcement Learning

In a FIS the rules governing the system are described linguistically using terms of natural language, and then they are transformed into fuzzy rules. These linguistic descriptions are constructed subjectively according to prior knowledge of the system, making the process highly dependent on the expert's knowledge. If the expert's knowledge about the system is faulty or the dynamics of the plant are either unknown or are too complex to be solved analytically. A learning method to fine-tune the parameters of the FIS will be needed in order to achieve optimality.

As described before, a plausible solution for the above described problem is the use of supervised learning, which uses input-output data in the development of the fuzzy model. When these input-output training data sets are available and reliable, supervised learning has proved to be more efficient than reinforcement learning (Anderson, 1986, Barto and Jordan, 1987). However, this is not always the case, in the supervised approach; the fuzzy rules are constructed based on the input-

output data, which can be faulty, damaged or noisy so that the obtained rules may not be reliable. Additionally the system might require the selection of control actions whose consequences emerge over uncertain periods of time for which input-output training data sets are not readily available. In this case, the use of reinforcement learning techniques are more appropriate than supervised learning (Berenji and Khedkar, 1992).

Recent studies have proved that reinforcement learning can be used to fine-tune fuzzy logic controllers successfully; either for structure identification as in (Lin and Xu, 2006, Lin and Lee, 1994) or for parameter identification like the Generalized Approximate-Reasoning-Based Intelligent Controller (GARIC) presented by (Berenji and Khedkar, 1992, Berenji and Khedkar, 1998) and the Generalized Reinforcement Learning Fuzzy Controller (GRLFC) developed by (Mustapha and Lachiver, 2000).

4.7 Conclusions

The state of uncertainty is defined for situation where the perceived information or the outcome of an action cannot be clearly defined due to factors as incomplete information or ambiguous contradicting information. Therefore the development of methods to deal with this issue is important so that a more accurate prediction of future outcomes can be used to select present actions.

Several theories for handling uncertainty have been developed for different types of uncertainties. The best developed and most often used is probability theory, which quantifies the likelihood of an imperfect (noise or otherwise corrupted) information to be a correct one.

Fuzzy sets, unlike probability and related theories, provide a mean to represent vagueness, instead of quantifying the likelihood of an event. These sets generalize

the concept of a truth value, assigning a membership value instead, to a set of events. The basic assumption is that the data is vague, due to set-valued information, and fuzzy sets can serve as a compressed description of imprecise, generally contradicting pieces of such information.

Several authors combined probabilities and fuzzy sets, the attempts range from using probability theory related methods on fuzzy models to defining concepts of probability theory on fuzzy sets.

4.8 Summary

This chapter explored the unavoidable issue of uncertainty, explaining its different sources and the existing methods to deal with this problem. Two methods for dealing with different aspects of uncertainty have being described; fuzzy logic and probabilistic theory. Finally a framework combining these two methods was described.

In the next chapter, the development of a method using the described probabilistic fuzzy logic theory in combination with reinforcement learning will be described.

Chapter 5

GPFRL Generalized Probabilistic Fuzzy-Reinforcement Learning

5.1 Introduction

The previous chapters described two major issues in the design of fuzzy logic systems. For one side fuzzy logic systems (FLS) are only able to handle fuzziness, non-statistical and non-stochastic uncertainties but, do not possess the capability to handle statistical and stochastic uncertainties. On the other hand, if the extraction of the expert's knowledge is difficult, the decision rules have complex structures and the number of variables necessary to solve the control task is excessively large. The design of the FLS becomes too complex. And the risk of achieving sub-optimal performance is high.

The proposed solution merges 3 different paradigms in order to deal with the described issues. First a FIS is combined with probabilistic theory as described in chapter 4 into a probabilistic fuzzy logic PFL. This combination enhances the uncertainty handling capabilities of fuzzy logic system or probability alone.

Despite of the added benefit of combining these two paradigms, doing this adds complexity to the system, making it more difficult for a human expert to configure or even to understand, so that an optimal configuration becomes unlikely. Second the introduction of reinforcement learning into the PFL ensures an optimal configuration of the system.

The present chapter presents such an algorithm, combining fuzzy logic systems with probabilistic theory and reinforcement learning into a Generalized Probabilistic Fuzzy-Reinforcement Learning (GPFRL) algorithm, able to handle all the above mentioned issues.

In this novel algorithm the reinforcement learning stage is used to find probabilities of success of each fuzzy system state. This algorithm features a novel value function update algorithm that incorporates a probabilistic term, so that the uncertainty is considered in the learning stage. As it will be seen in the results of the experiments presented in chapter 6, this yielded to improved learning speeds, faster convergence and robustness in light of input and output uncertainties.

The rest of this chapter is organized as follows; section two describes the architecture of our proposed GPFRL, section three describes the fuzzyfication and defuzzyfication method used in our proposed approach. Section four details the GPFRL learning process. The description of our proposed approach is given in section five and finally section six summarizes the contents of the present chapter.

5.2 Structure

The proposed algorithm uses an Actor-Critic method. This kind of methods are a special case of temporal difference (TD) methods (Barto et al., 1983) formed by two structures. The Actor is a separate memory structure to explicitly represent

the control policy which is independent of the value function, which its function is to select the best control actions. The Critic has the task to estimate the value function and it is called that way because it criticizes the control actions made by the actor. TD error depends also on the reward signal obtained from environment as a result of the control action. Figure 5.1 shows the actor-critic configuration; where r represent the reward signal, $\bar{r}$ is the internal enhanced reinforcement signal and a is the selected action for the current system state.

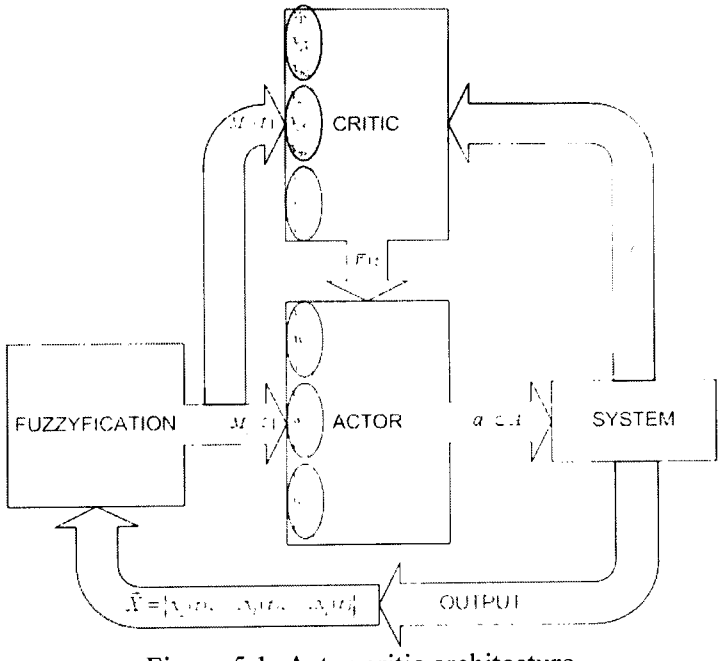

Figure 5.1: Actor-critic architecture.

5.3 Probabilistic Fuzzy Inference

After an action $a_k \in A = \{a_1, a_2, ..., a_n\}$ is executed by the system, the learning agent performs a new observation of the system. This observation is composed by a vector of inputs that inform the agent about external or internal conditions that

can have a direct impact on the outcome of a selected action. These inputs are then processed using Gaussian membership functions according to

$$\begin{aligned}
&\text{Left shoulder:} \quad \mu_i^L(t) = \begin{cases} 1 & \text{if } x_i \le c^L \\ e^{-\frac{1}{2}\left(\frac{x_i(t)-c^L}{\sigma^L}\right)^2} & \text{otherwise} \end{cases} \\
&\text{Centre MFs:} \quad \mu_i^C(t) = e^{-\frac{1}{2}\left(\frac{x_i(t)-c^C}{\sigma^C}\right)^2} \\
&\text{Right shoulder:} \quad \mu_i^R(t) = \begin{cases} e^{-\frac{1}{2}\left(\frac{x_i(t)-c^R}{\sigma^R}\right)^2} & \text{if } x_i \le c^R \\ 1 & \text{otherwise} \end{cases}
\end{aligned} \tag{5.1}$$

where $\mu_i^{\{L,C,R\}}$ is the firing strength of input x_i, $i = \{1,2,...,I\}$ is the input number, L, C and R specify the type of membership function used to evaluate each input; x_i is the normalized value of input i; $c^{\{L,C,R\}}$ is the centre value of the Gaussian membership function and $\sigma^{\{L,C,R\}}$ is the standard deviation for the corresponding membership function.

These success probabilities ρ_{jk} are the normalization of the s-shaped weights of the actor, evaluated at time step t and are defined as

$$\rho_{jk}(t) = \frac{S\left[w_{jk}(t)\right]}{\sum_{j=1}^{m} S\left[w_{jk}(t)\right]} \tag{5.2}$$

where S is an s-shaped function given by

$$S\left[w_{jk}(t)\right] = \frac{1}{1+e^{-w_{jk}(t)}} \tag{5.3}$$

and $w_{jk}(t)$ is a real-valued weight that maps rule j with action k at a time t.

The total probability of success of performing action a_k considers the effect of all individual probabilities and combines them using a weighted average where, M_j are all the consequents of the rules and $P_k(t)$, is the probability of success of executing action a_k at time step t. The final action to be selected will be a weighted combination of all actions and their probabilities.

$$P_k(t) = \frac{\sum_{j=1}^{m} \mathrm{M}_j(t) \cdot \rho_{jk}(t)}{\sum_{j=1}^{m} \mathrm{M}_j(t)} \tag{5.4}$$

In (5.4) $M_j(t)$ is the T-norm and is implemented by the product:

$$M_j(t) = \prod_{i=1}^{I} \mu_{ij}(x_i) \tag{5.5}$$

where μ_{ij} is the j^{th} membership function of rule R_i. Furthermore, fuzzy basis functions are defined as follows:

$$M_j = \frac{M_j(x)}{\sum_{j=1}^{m} M_j(x)} \tag{5.6}$$

In most reinforcement learning implementation, there is an issue concerning the trade-off between "exploration" and "exploitation" (Sutton and Barto, 1998). It is the balance between trusting that the information obtained so far is sufficient to make the right decision (exploitation) and trying to get more information so that better decisions can be made (exploration). For example, when a robot faces an

unknown environment, it has to first explore the environment to acquire some knowledge about the environment. The experience acquired must also be used (exploited) for action selection to maximize the rewards (Kantardzie, 2002).

Choosing an action merely considering $P_k(t)$ will lead to a exploiting behaviour. In order to create a balance, (Barto et al., 1983) suggested the addition of a noise signal with mean zero and a Gaussian distribution. The use of this signal will force the system into an explorative behaviour where different than optimum actions are selected for all states; thus a more accurate input-output mapping is created at the cost of learning speed. In order to maximize both, accuracy and learning speed an enhanced noise signal is proposed. This new signal is generated by a stochastic noise generator defined in (5.7).

$$\eta_k = N(0, \sigma_k) \tag{5.7}$$

Where N is a random number generator function with a Gaussian distribution, mean zero and a standard deviation σ_k which is defined as

$$\sigma_k = \frac{1}{1 + e^{[-2 p_k(t)]}} \tag{5.8}$$

The stochastic noise generator uses the prediction of eventual reinforcement, $p_k(t)$ shown in (5.11), as a damping factor in order to compute a new standard deviation. The result is a noise signal which is more influential at the beginning of the runs, boosting exploration, but quickly becomes less influential as the agent learns, leaving the system with its default exploitation behaviour.

5.4 Reinforcement Learning Process

The learning process of a GPFRL is based on an Actor-Critic reinforcement learning scheme, where the actor learns the policy function and the Critic learns the value function using the TD method simultaneously. This makes possible to focus on on-line performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

Formally, the basic RL model consists of:

- A set of environment state observations O.
- A set of actions A.
- A set of scalar "rewards" r.

At each discrete time step t, the environment generates an observation $o(t)$ of the environment and acquires a set of inputs $x_t \in X$, then the agent performs and action which is the result of the weighted combination of all the possible actions $a_k \in A$, where A is a discrete set of actions. The action and observation events occur in sequence, $o(t), a(t), o(t+1), a(t+1),...$ which will be called experience. In this sequence, each event depends only on those preceding it.

RL typically requires an unambiguous representation of states and actions and the existence of a scalar reward function. For a given state, the most traditional of these implementations would take an action, observe a reward, update the value function and select, as the new control output, the action with the highest expected value (probability) in each state (for a greedy policy evaluation). The updating of the value function is repeated until convergence is achieved. This procedure is usually summarized under policy improvement iterations.

The parameter learning of the GPFRL system includes two parts: the Actor parameter learning and the Critic parameter learning. One feature of the Actor–Critic learning is that the learning of these two parameters is executed simultaneously.

Given a performance measurement $Q(t)$ and a minimum desirable performance $Q_{\min}$ we define the external reinforcement signal r as

$$r = \begin{cases} 0, \ \forall\, Q(t) \geq Q_{\min} > 0 \\ -1, \ \forall\, 0 \leq Q(t) < Q_{\min} \end{cases} \tag{5.9}$$

The internal reinforcement, $\bar{r}$, expressed in (5.10), is calculated using the temporal difference of the value function between successive time steps and the external reinforcement.

$$\bar{r}_k(t) = r(t) + \gamma p_k(t) - p_k(t-1) \tag{5.10}$$

where γ is the discount factor used to determine the proportion of the delay to the future rewards and the value function $p_k(t)$ is the prediction of eventual reinforcement for action a_k and is defined as

$$p_k(t) = \sum_{j=1}^{m} \mathrm{M}_j(t) \cdot v_{jk}(t) \tag{5.11}$$

where v_{jk} is the Critic weight of the j^{th} rule, described by (5.13).

5.4.1 Critic Learning

The goal of reinforcement learning is to adjust correlated parameters in order to maximize the cumulative sum of the future rewards. The role of the Critic is to

estimate the value function of the policy followed by the Actor. The TD error is the temporal difference of the value function between successive states. The goal of the learning agent is to train the Critic to minimize the squared TD error $E_k(t)$ described as

$$E_k(t) = \frac{1}{2}\overline{r}_k^2(t) \tag{5.12}$$

We use the gradient descent method in order to find the updating rule for the weights v_{jk} of the Critic (Baird and Moore, 1999)

$$v_{jk}(t+1) = v_{jk}(t) - \beta \frac{\partial E_k(t)}{\partial v_{jk}(t)} \tag{5.13}$$

where β is the learning rate.

Rewriting (5.13) using the chain rule:

$$v_{jk}(t+1) = v_{jk}(t) - \beta \frac{\partial E_k(t)}{\partial \overline{r}_k(t)} \cdot \frac{\partial \overline{r}_k(t)}{\partial p_k(t)} \cdot \frac{\partial p_k(t)}{\partial v_{jk}(t)} \tag{5.14}$$

$$\frac{\partial E_k(t)}{\partial \overline{r}_k(t)} = \overline{r}_k(t) \tag{5.15}$$

$$\frac{\partial \overline{r}_k(t)}{\partial p_k(t)} = \gamma \tag{5.16}$$

$$\frac{\partial p_k(t)}{\partial v_{jk}(t)} = M_j(t) \tag{5.17}$$

$$v_{jk}(t+1) = v_{jk}(t) - \beta\gamma\overline{r}_k(t) \cdot \mathrm{M}_j(t) \tag{5.18}$$

$$v_{jk}(t+1) = v_{jk}(t) - \beta' \bar{r}_k(t) \cdot \mathrm{M}_j(t) \tag{5.19}$$

In (5.19) β' is the new Critic learning rate.

5.4.2 Actor Learning

The main goal of the Actor is to find a mapping between the input and the output of the system that maximizes the performance of the system by maximizing the total expected reward. We can express the Actor value function $\lambda_j(t)$ according to

$$\lambda_k(t) = \sum_{j=1}^{m} \mathrm{M}_j(t) \cdot \rho_{jk}(t) \tag{5.20}$$

Equation (5.20) represents a mapping from an m dimensional input state derived from $x(t) \in \mathbb{R}^m$ to an n dimensional state $a_k \in \mathbb{R}^n$.

Then we can express the performance function $F_k(t)$ as

$$F_k(t) = \lambda_k(t) - \lambda_k(t-1) \tag{5.21}$$

Using the gradient descent method we define the actor weight updating rule as

$$w_{jk}(t+1) = w_{jk}(t) - \alpha \frac{\partial F_k(t)}{\partial w_{jk}(t)} \tag{5.22}$$

where α is a positive constant that specifies the learning rate of the weight w_{jk}. Then using the chain rule:

$$\frac{\partial F_k(t)}{\partial w_{jk}(t)} = \frac{\partial F_k(t)}{\partial \lambda_k(t)} \cdot \frac{\partial \lambda_k(t)}{\partial \rho_{jk}(t)} \cdot \frac{\partial \rho_{jk}(t)}{\partial w_{jk}(t)} \tag{5.23}$$

$$\frac{\partial F_k(t)}{\partial \lambda_k(t)} = 1 \tag{5.24}$$

$$\frac{\partial \lambda_k(t)}{\partial \rho_{jk}(t)} = M_j(t) \tag{5.25}$$

$$\frac{\partial \rho_{jk}(t)}{\partial w_{jk}(t)} = \frac{\partial \rho_{jk}(t)}{\partial S[w_{jk}(t)]} \cdot \frac{\partial S[w_{jk}(t)]}{\partial w_{jk}(t)} \tag{5.26}$$

$$\frac{\partial \rho_{jk}(t)}{\partial w_{jk}(t)} = \frac{1}{\rho_{jk}(t)} \cdot \sum_{j=1}^{m} S[w_{jk}(t)] \cdot e^{-w_{jk}(t)} \tag{5.27}$$

$$\frac{\partial F_k(t)}{\partial w_{jk}(t)} = \mathrm{M}_j(t) \cdot \frac{1}{\rho_{jk}(t)} \cdot \sum_{j=1}^{m} S[w_{jk}(t)] \cdot e^{-w_{jk}(t)} \tag{5.28}$$

Hence, we obtain

$$w_{jk}(t+1) = w_{jk}(t) - \alpha \cdot \mathrm{M}_j(t) \cdot \frac{1}{\rho_{jk}(t)} \cdot \sum_{j=1}^{m} S[w_{jk}(t)] \cdot e^{-w_{jk}(t)} \tag{5.29}$$

Equation (5.29) represents the generalized weight update rule.

5.5 Algorithm Description

In Figure 5.2 the GPFRL algorithm flowchart is presented. The overall structure is equivalent to the one of the actor critic methods. The flowchart of the left shows the sequence of the main program, which can be configured for many executions or "runs". The purpose of this is to average the resulting values (such as number

of trials for learning, rewards, etc.), in order to obtain more representative results. The flowchart on the right shows the main GPFRL algorithm.

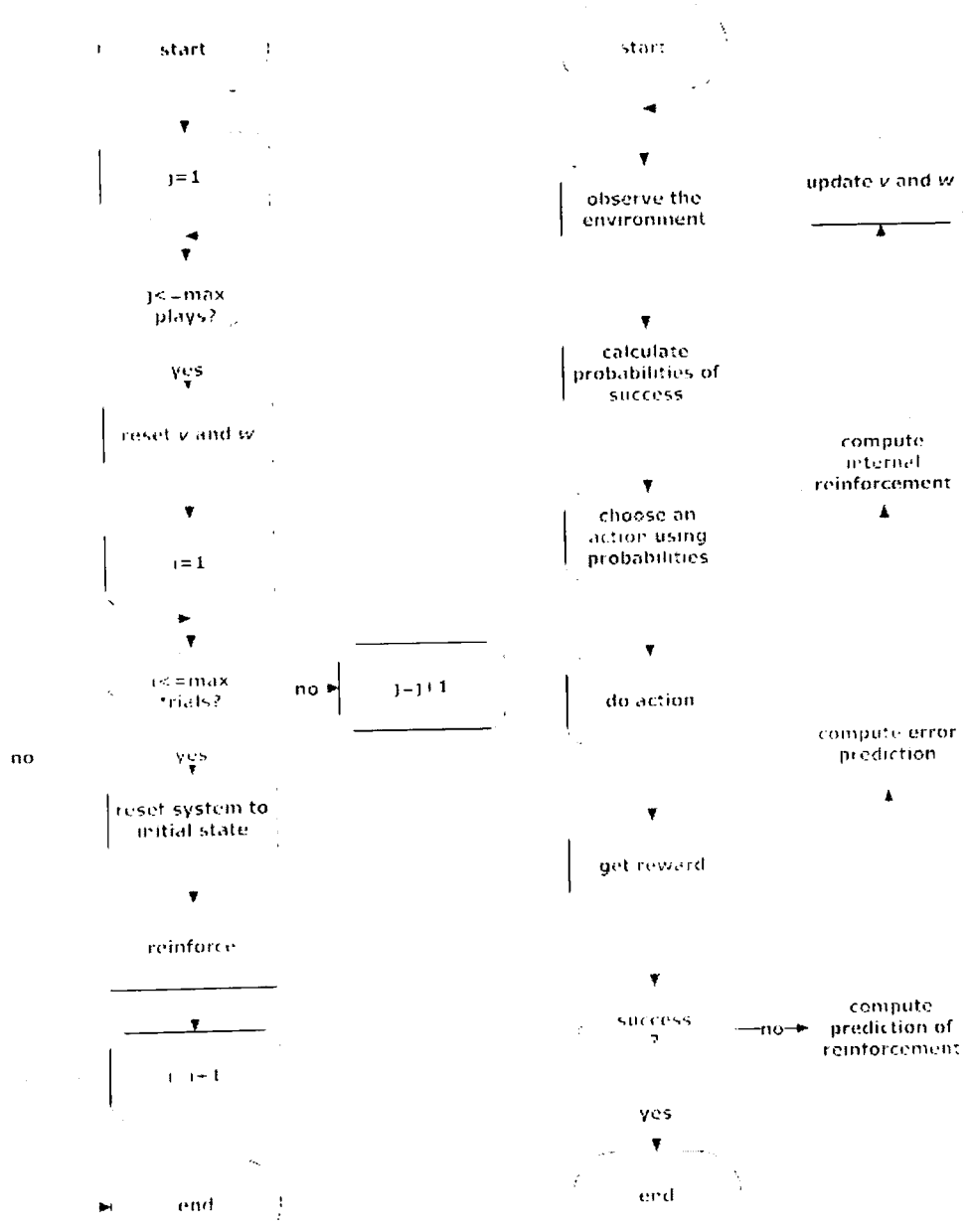

Figure 5.2: GPFRL algorithm flowchart.

Table 5.1 shows the proposed GPFRL pseudo code.

TABLE 5.1 GPFRL ALGORITHM

Algorithm

Initialize v_{jk} and w_{jk}

Repeat (for each episode):

Initialize s

Choose a from s using policy derived from P_k

Repeat (for each step of episode):

$$\rho_{jk}(t) = \frac{S\left[w_{jk}(t)\right]}{\sum_{j=1}^{m} S\left[w_{jk}(t)\right]}$$

$$P_k(t) = \frac{\sum_{j=1}^{m} \mathrm{M}_j(t) \cdot \rho_{jk}(t)}{\sum_{j=1}^{m} \mathrm{M}_j(t)}$$

$$\sigma_k = \frac{2}{1 + e^{\left[-2P_k(t)\right]}} - 1$$

$$\eta_k = N(0, \sigma_k)$$

$$a' = \max_k \left[P_k + \eta_k\right]$$

Choose a' from s' using policy derived from $w_{jk}(t)$

Compute $r(t)$ from performance index

$$p_k(t) = \sum_{j=1}^{m} \mathrm{M}_j(t) \cdot v_{jk}(t)$$

$$\overline{r}_k(t) = r(t) + \gamma p_k(t) - p_k(t-1)$$

$$v_{jk}(t+1) = v_{jk}(t) - \beta' \overline{r}_k(t) \cdot \mathrm{M}_j(t)$$

$$w_{jk}(t+1) = w_{jk}(t) - \alpha \cdot \mathrm{M}_j(t) \cdot \frac{1}{\rho_{jk}(t)} \cdot \sum_{j=1}^{m} S\left[w_{jk}(t)\right] \cdot e^{-w_{jk}(t)}$$

Get inputs

Fuzzyfy inputs compute $\mu_i(t)$

$$M_j(t) = F\left(\mu_i(t)\right)$$

Until s is terminal

5.6 Discussion

This chapter presented the development of a probabilistic fuzzy-reinforcement learning algorithm. The fusion of these two paradigms has been the focus of researchers for many years for the unique advantages of these two methods. When combined with fuzzy logic systems, the RL task can two tasks, to fine tune the controller either performing structure identification, as studied by (Lin and Xu, 2006, Lin and Lee, 1993, Wang et al., 2007) or as a parameter identification like in: (Berenji and Khedkar, 1992, Lin and Lee, 1993, Wang et al., 2007). The work presented in this dissertation concentrates on the latter issue. In similar way the combination of fuzzy logic with probabilistic theory have been envision to be an excellent way to manage information in the presence of different kinds of uncertainties. It is to the best of the authors understanding that no similar method have combined these three paradigms, at least not in the same synergistic way as it is presented in this work.

Nevertheless, there is a great amount of research concerning different methods to improve or combine reinforcement learning in order to improve/expand its capabilities. The original work can be found in (Barto et al., 1983) where a box system was used for the purpose of describing a system state based on its input variables, which the agent was able to use to decide an appropriate action to take. The previously described system uses a discrete input, where the system was described as a number which represented the corresponding input state. A better approach will consider a continuous system characterization, like in (Lin, 1995) whose algorithm is based on the AHC but with the addition of continuous inputs, by ways of a two layers neural network but show poor performance in its learning time. Berenji introduced GARIC (generalized ARIC) in (Berenji and Khedkar, 1992) the use of output stage and the use of structure learning in his architectures, further reducing the learning time. Another interesting approach was proposed by

Lee (Lee, 1991) which uses neural networks and approximate reasoning theory, but an important drawback in Lee's architecture is its inability to work as a standalone controller (without the learning structure). C. T. Lin's developed two approaches: a reinforcement neural-network-based fuzzy logic control system (RNN-FLCS) (Lin and Xu, 2006) and a Reinforcement Neural Fuzzy Control Network (RNFCN) (Lin, 1995) both were endowed with structure and parameter learning capabilities. L ater, C. J. Lin developed FA LCON-RL (reinforcement learning strategy based on fuzzy adaptive learning control network) method (Lin and Lee, 1993), Jouffe's FACL (fuzzy AC learning) method (Jouffe, 1998), C. K. Lin's RLAFC (reinforcement learning adaptive fuzzy controller) method (Lin, 2003), Wang's FACRLN (fuzzy Actor–Critic reinforcement learning network) method (Wang et al., 2007). Zarandi proposed a Generalized Reinforcement Learning Fuzzy Controller (GRLFC) method (Zarandi et al., 2008) that was able to handle vagueness on its inputs but overlooked the handling of ambiguity, an important component of uncertainty, although his approach showed a fast learning, this can be due to the reduced testing time (33000 time steps compared to the standard 500000) used in its trials, also the structure of his system became complex by using 2 independent FISs. Several other (mainly model-free) fuzzy RL algorithms have been proposed, being mostly Q-learning (Jouffe, 1998, Lin, 2003, Almeida and Kaymak, 2009, Lin and Lin, 1996) or acto"-critic techniques (Jouffe, 1998, Lin, 2003). Most of the cited algorithms present many different improvements to RL, but they fail to provide a way of handling a wide spectrum of uncertainty, in the same proportion it is done by using a probabilistic fuzzy system.

In the last few years some similar studies merged 2 or more of the paradigms proposed in this dissertation; in 2000 (Strens, 2000) uses the idea that the uncertainty in the environment's underlying Markov Decision Process (MDP) model can be encoded with a probability distribution. He initially constructs a

distribution over all possible MDP models, referred to as hypotheses, and keeps updating this distribution after observing evidence at each time step. This distribution is referred to as the agent's belief. The policy at any time is given by sampling a hypothesis from the current belief and determining the optimal policy for that hypothesis. With time, the peak of the distribution shifts towards the true MDP and so the policy converges to the optimal one. This approach implicitly optimizes the tradeoffs between exploration and expectation in the sense that agents will naturally tend to explore more often in the beginning of the learning process and less when nearing convergence.

Dearden et al. propose a similar framework in their work on Bayesian Q-learning (Dearden et al., 1998). However, instead of uncertainty in the environment, they model uncertainty in the value of information associated with performing an exploratory action. While this does not learn the underlying model of the environment, it provides a more structured approach to measuring the tradeoffs between exploration and exploitation. Namely, the agent can directly compare the value of expected future rewards to the value of gaining exploratory information and make a choice based on this comparison. Through the course of learning the uncertainty is reduced and the selected actions converge to the optimal policy.

Some years later, a better solution was proposed by (Poupart et al., 2006), where they model the environment and the uncertainty in it with a Partially Observable MDP (POMDP). A POMDP is an extension of an MDP where the state space is not fully observable but instead can be reasoned about through an explicitly defined set of observations. In this dissertation, the underlying MDP model of the environment is considered a part of the state space (i.e., the partially observable part) and is learned through the course of acting. (Poupart et al., 2006) show that in this case the optimal value function is the upper envelope of a set of multivariate polynomials and develop an algorithm that exploits this fact to compute an optimal policy offline. Their algorithm is practical because it allows

online learning with minimal computation overhead (only belief monitoring is performed) and at the same time maximizes the expected total reward. The approach also optimizes exploration and exploitation directly, since the action selection process implicitly takes into account how the belief will change as a result of the selected action.

Fuzzy Inference Systems (FIS) can be used to facilitate generalization in the state space and to generate continuous actions (Jouffe, 1998). In (Jouffe, 1998), the state-space coding is realized by the input variable's fuzzy set. The selection of more than one fuzzy state at any point of time results in smooth transition between a state and its neighbours and consequently, smooth changes between actions. Additionally, reinforcement learning has been used in conjunction with fuzzy inference systems, basically for the task of optimization by using two different methods:

- Methods based on policy iteration, driving to Actor-Critic architectures (Berenji and Khedkar, 1992).

- Methods based on v alue it eration and generalize Q-Learning (Berenji, 1996, Glorennec and Jouffe, 1997, Glorennec, 2000). In (Glorennec, 2000), a Q-Learning algorithm was used for the optimization of a zero order Takagi-Sugeno fuzzy inference system, with a constant conclusions.

In the rest of this dissertation, it is considered that a Takagi-Sugeno fuzzy inference system and continuous state and action spaces are used. The FIS structure is fixed a priori by the user and the fuzzy sets for the inputs and output are supposed fixed. Our approach, consist in determining the optimal conclusions of the fuzzy inference system.

5.7 Summary

This chapter described a novel general method for decision making and control systems that combines probabilistic fuzzy logic theory with reinforcement learning. The presented method uses the actor-critic structure, were a critic is composed by a state-value function based on temporal difference learning, which is used to criticize the actions of the actor (action/decision maker). The actor task is to use state information to decide an appropriate action to take, based on probabilities of success. These probabilities of success are found by sequential interactions of the agent and its environment by using our proposed reinforcement learning algorithm. This chapter concludes presenting a discussion describing several other methods found in the literature, that use reinforcement learning.

The next chapter presents four specific experiments performed, in order to test different aspects of our proposed method, as convergence, learning speed, and stability and robustness under uncertainty. Whilst comparing it with other methods under similar conditions.

Chapter 6

Experiments

6.1 Introduction

In this chapter, we consider a number of examples regarding our proposed reinforcement learning approach. In section two, we consider a random walk problem under a grid world in order to evaluate the convergence of the learning algorithm while comparing its performance with two classic temporal difference algorithms, SARSA and Q-Learning. In section three, we used a classic benchmark for learning algorithms, the control of a simulated cart-pole system; this was used to compare the performance of our proposed algorithm with other reinforcement learning approaches. Section four describes the control of a loaded DC motor whilst in section five; our algorithm was implemented in a real mobile robot for solving the navigation problem of obstacle avoidance. Finally, section six summarizes the contents of this chapter.

6.2 Decision Making Experiments

6.2.1 Random Walk Problem

A random walk problem is a mathematical formalisation of a trajectory that consists of taking successive random steps. The results of random walk analysis have been applied to computer science, physics, ecology, economics, psychology, and a number of other fields as a fundamental model for random processes in time.

The purpose of this experiment is to provide a mean to evaluate the updating rule, the convergence, and stability of our proposed GPFRL and compare it with other two well-known TD learning methods, SARSA and Q-Learning. Figure 6.1 shows an example of a 5x5-grid world. In Figure 6.1, the learning agents start from the upper right corner marked as "S" (start state) and the task of the learning agent is to find the shortest path from the state "S" and the state "G" (goal state) in the shortest possible time. In the above example there are many possible solutions, two of them are marked with a red line and a blue line in Figure 6.1. In any case, the shortest path consists in six steps.

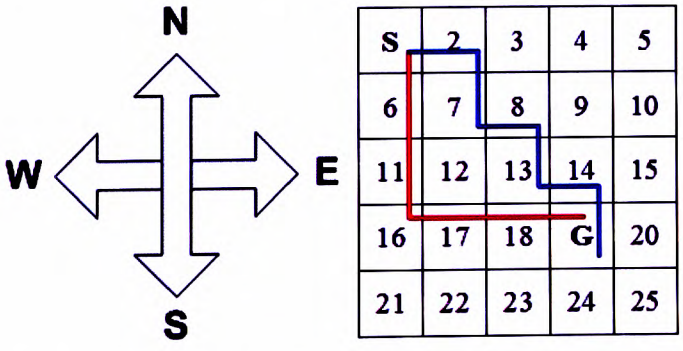

Figure 6.1: A 5x5 dimension grid world for a total of 25 states.

6.2.1.1 Grid World exploration

In this experiment the learning problem is to learn how to choose the correct direction to take for every state, in order for the learning agent to reach the goal state with the fewest amounts of steps. To accomplish this task the learning agent will explore every option for every state, evaluate the long term outcome and then update the value functions of every state, so that the next time this state is visited, the learning agent will be able to select the action with the highest probability of success, in other words, that will allow the agent to reach the goal state with the smallest number of steps.

It becomes clear that the probabilities of success for each of the choices for every given state will diverge from their initial values towards their real values in direct proportion with the number of times the state is visited. Therefore in order to find accurate probabilities for every action for every state, all states must be visited as many times as possible. The main disadvantage of this exploratory behaviour is that is time consuming. A direct solution will be to eliminate the exploratory behaviour of the agent so that it will become "greedy" and will choose always the actions that are considered to be optimal. The danger with this approach is that there will be many states "unvisited", therefore the probabilities of success of the possible actions for each one of this states will be unknown.

In order to achieve a balance between exploration-exploitation a stochastic signal is introduced to the action selection process. Considering that there is only one learning agent for every action and every learning agent assigns a probability value to every action according to (6.1)

$$P'_k = P_k + \eta_k \tag{6.1}$$

Where η_k was defined in (5.7) and (5.8). If there is only one action with a probability value equal to $\max(P'_k)$ and this probability value is equal or greater than a predefined threshold value of $\varepsilon \mid \varepsilon \in \{0,1\}$, then the action is selected. Otherwise, an action $a \in s$ is selected randomly. The value of ε can be defined by the programmer in order to introduce a more explorative behaviour or a more exploitative one.

6.2.1.2 Deterministic Environment

6.2.1.2.1 System Description

This grid world uses a rectangular grid to illustrate value functions for a simple finite MDP. The cells of the grid correspond to the states of the environment. At each cell, four actions are possible: North, South, East, and West, which deterministically cause the agent to move one cell in the respective direction in the grid.

All episodes start in the upper left corner state, or state "1", $S(1, 1)$, and proceed either North, South, East or West by one state on each step, with equal probability. The task is to find an optimal policy which will let the agent move from $S(1, 1)$ to $G(m_g, n_g)$ with minimized cost (number of moving steps).

For the present example, we used a 10 by 10 grid world, for a total of 100 possible states, with the initial position of the learning agent set at (1, 1) (state "1") and the goal state was fixed at (5, 6) (state "46"). A state order can be assigned according to eq. (6.2), where m and n are the vertical and horizontal axis positions of the grid and $n_{\max}$ is the total number of rows of the grid world.

$$n_{\max} \times (m-1) + n \tag{6.2}$$

For the SARSA and Q-Learning algorithms the agent receives a reward of zero, "0" for every step that is not the goal and a reward of one "1" when it reaches the goal and then ends this episode. In our approach there are four learning agents, where each one learns the probabilities of success of going in each one of the four directions. So for every step that the agent takes that result with the agent being in a state closer to the goal state, it receives a reward of one, "1"; in the same way for every step it takes in a direction that is not towards the goal, it receives a punishment of "-1". Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1. Other actions result in a reward of 0.

TABLE 6.1 PARAMETERS USED FOR THE RANDOM WALK EXPERIMENT.

Parameter	SARSA	Q-Learning	GPFRL
α	0.4	0.4	0.003
β		-	0.005
γ	0.95	0.95	0.95
ε	0.01	0.01	0.01

The discount factor γ was set to 0.95 for all the algorithms that we have carried out in this example. For the action selection policy of TD algorithm, we use an ε-greedy policy ($\varepsilon = 0.01$).

The parameters used for these tests are shown in Table 6.1. In addition, the corresponding code is shown in Appendix A. The described tests were completely coded and evaluated using MATLAB®.

6.2.1.2.2 Fixed start state

For this test, the system was set to execute 40 trials and the results were stored and averaged for 100 plays. Figure 6.2 shows the average number of steps the agent takes for reaching the goal state from a static starting position in the grid world. It can be observed that the system not only converges to an optimum faster than the SARSA and Q-Learning algorithms, but also that it does so with less

steps in the first trial and an overall fast convergence. Figure 6.3 shows a graphical comparison of the standard deviations for the three algorithms, it can be observed that the GPFRL and Q-Learning algorithm have a standard deviation near zero, whilst SARSA showed a standard deviation of 10 in average.

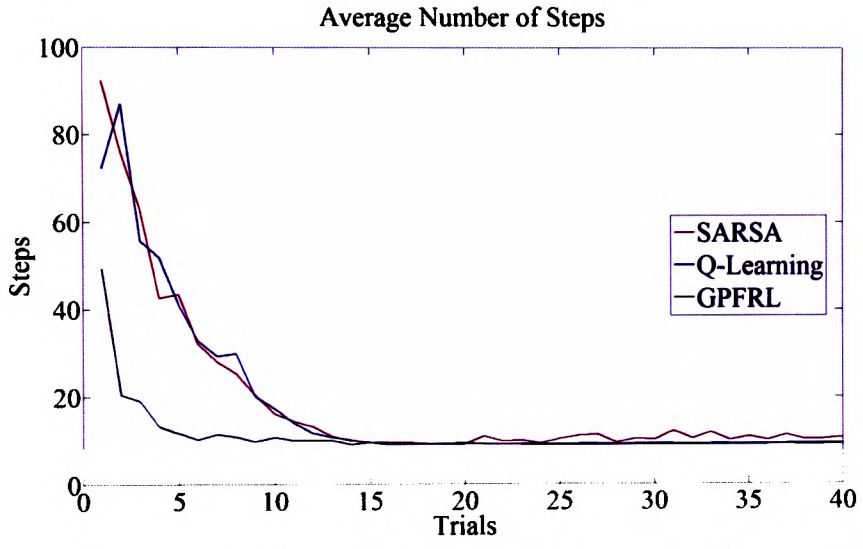

Figure 6.2: Static starting point learning rate comparison.

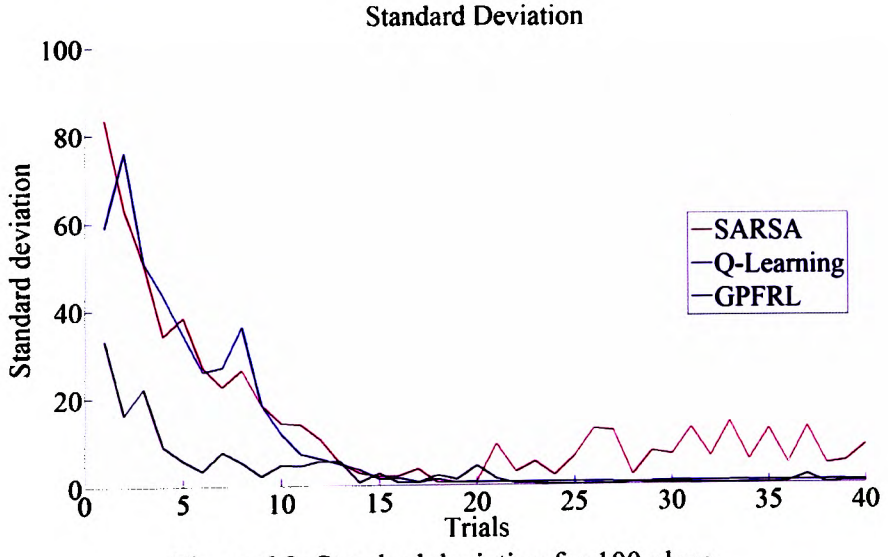

Figure 6.3: Standard deviation for 100 plays.

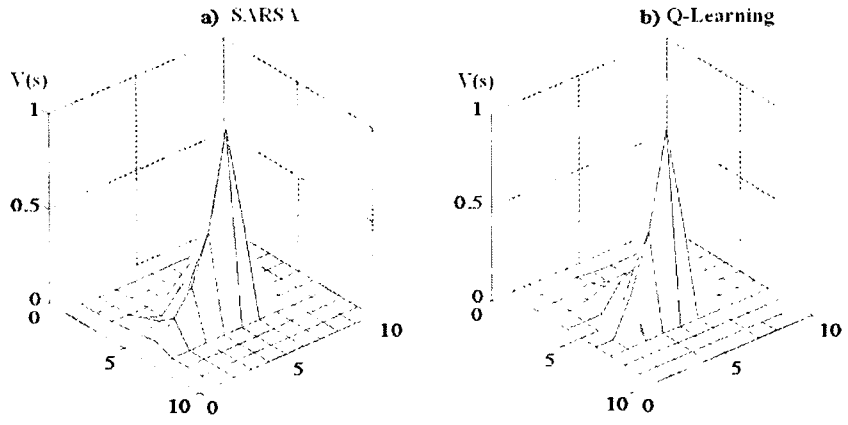

Figure 6.4: Static starting point utility value distributions for a) SARSA and b) Q-learning.

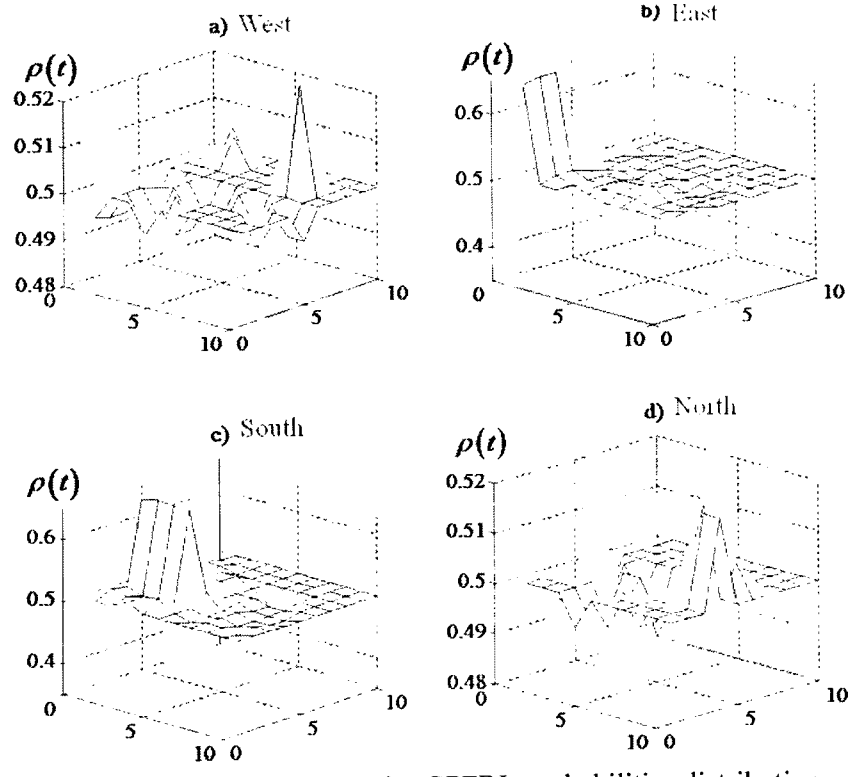

Figure 6.5: Static starting point GPFRL probabilities distribution.

Figure 6.4 shows in a) the utility value distribution using SARSA algorithm and in b), the utility value distribution using the Q-Learning algorithm. Both obtained

after 40 trials of the random walk experiment, starting from a static position towards a static goal.

Figure 6.5 shows the probabilities of success of taking direction a) West, b) East, c) South, and d) North; obtained after 40 trials of the random walk experiment using the GPFRL algorithm, again, starting from a static position towards a static goal.

In the temporal difference algorithms, SARSA and Q-learning, the goal state have a reward of "1", with this methods, the learning agent constructs a mapping "backwards" in other words, it assigns values to the states according to the proximity to the goal, in consequence, in order to start assigning values the learning agent must have visited the goal state at least once. For especially large, complicated (stochastic) environments this task can be very time consuming and inefficient.

The value assignment can be clearly visualized in Figure 6.4 where its hill shape indicate the relative value of the corresponding state, in relation with the value assigned to the goal state. In this case the learning agent follows a greedy policy (exploitation), where it will follow the path with the highest steep. This can also be noticed by observing the "flat" surrounding states to the hill, where no value have been assigned, indicating that either this states have not been visited or that the visit did not lead to a state with an assigned value. In an exploration mode, it can be expected all the states to be visited, and a value assigned to each.

Figure 6.5 shows a surface area generated by all the states and the probabilities associated to them. Each subplot corresponds to these probabilities as seen by each one of the four learning agents. In this graphs it can be observed a series of hills and dips, as a contrast with the SARSA and Q-Learning methods where only one hill could be devised. For the GPFRL graphs, the dips can be understood as a low probability of taking the corresponding direction from that state. This can be

contrasted by observing the complimentary graph, where the hills are replaced by dips and vice versa. This constitutes a richer source of options for the decision making process.

Another important thing to notice is the small probability values observed in Figure 6.5, since the system is configured to follow the greedy policy, the learning agent does not need a large difference of probabilities between to actions to select one, therefore it will always select the action with the highest probability numerical value. Forcing the learning agent to randomly select actions with small probability values, will lead to a exploration mode, where a larger surface is explored and states are visited more often, generating higher probability values

6.2.1.2.3 Random start state

For our second set of tests under the grid world, we used a random starting state for each trial and compared the results obtained. Also the number of trials was extended to 100 averaged for over 500 plays. Figure 6.6 shows the comparison of the average number of steps to reach the goal for the SARSA, Q-Learning and GPFRL; in Figure 6.8 we show the utility value distribution of the SARSA algorithm a), and the Q-Learning algorithm b). In contrast Figure 6.9 shows the probability distribution for the GPFRL algorithm. Figure 6.7 shows a much increased standard deviation for SARSA and Q-learning which did not approached zero even after many trials. It can also be seen in Figure 6.7 that the GPFRL have a low and steady standard deviation for all trials, which means that for every run, the Khepera III robot with the GPFRL followed the shortest path with a number of steps difference as large as the distance between the goal and the farthest-from-the-goal starting point.

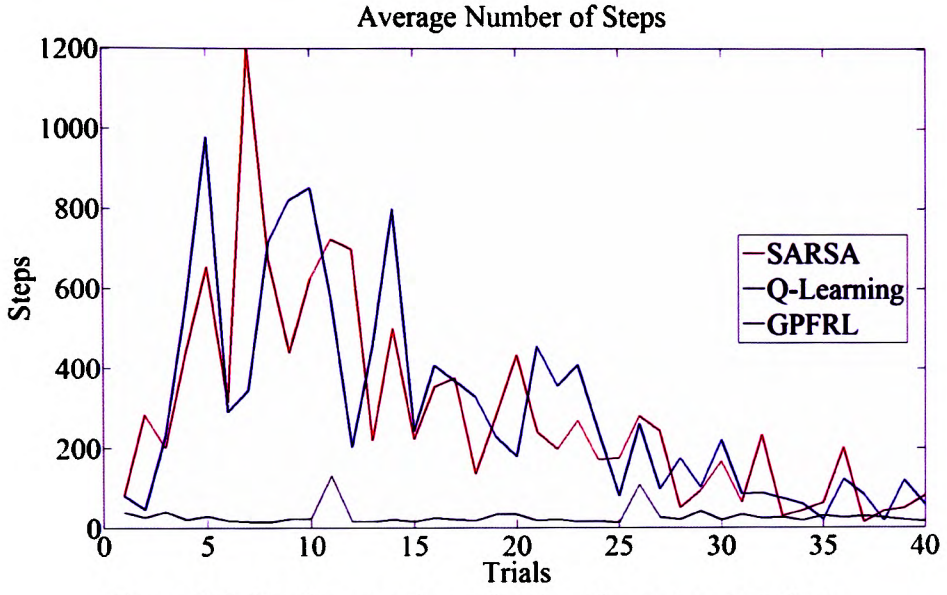

Figure 6.6: Random starting point, learning rate comparison.

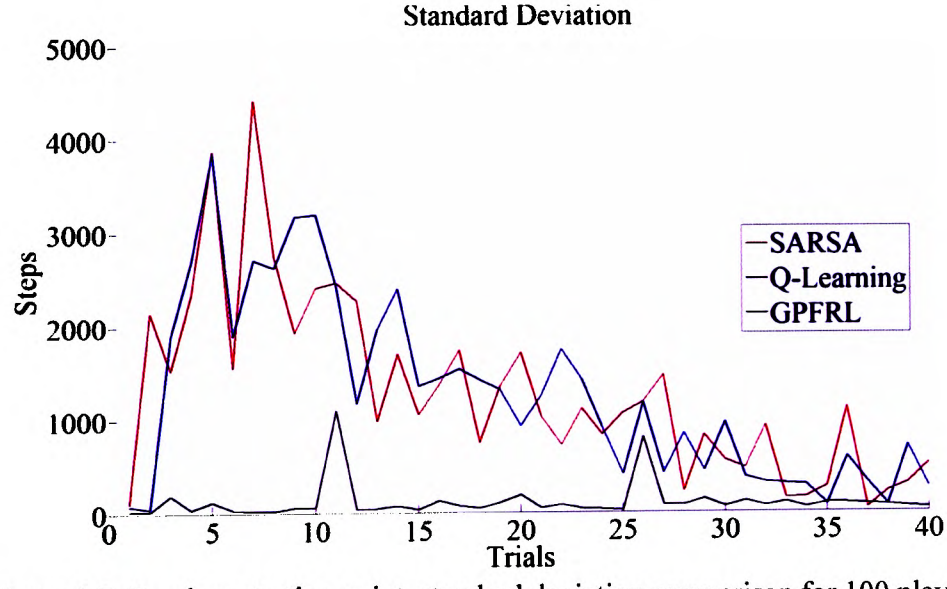

Figure 6.7: Random starting point, standard deviation comparison for 100 plays.

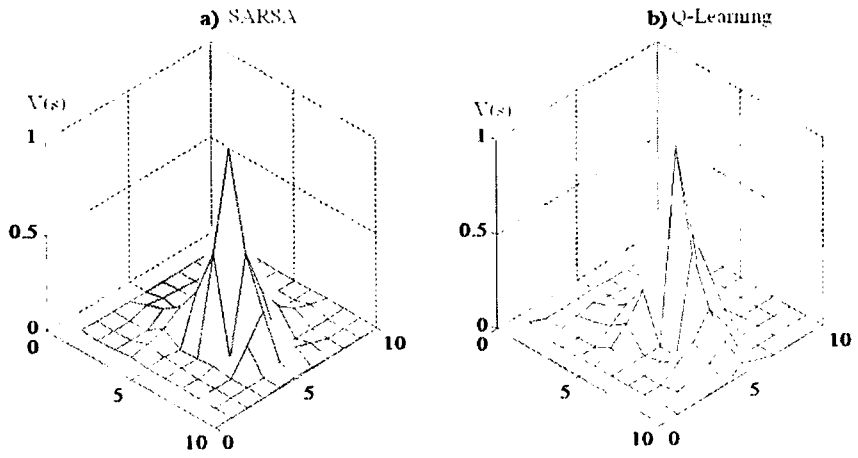

Figure 6.8: Random starting point SARSA and Q-Learning utility value distribution.

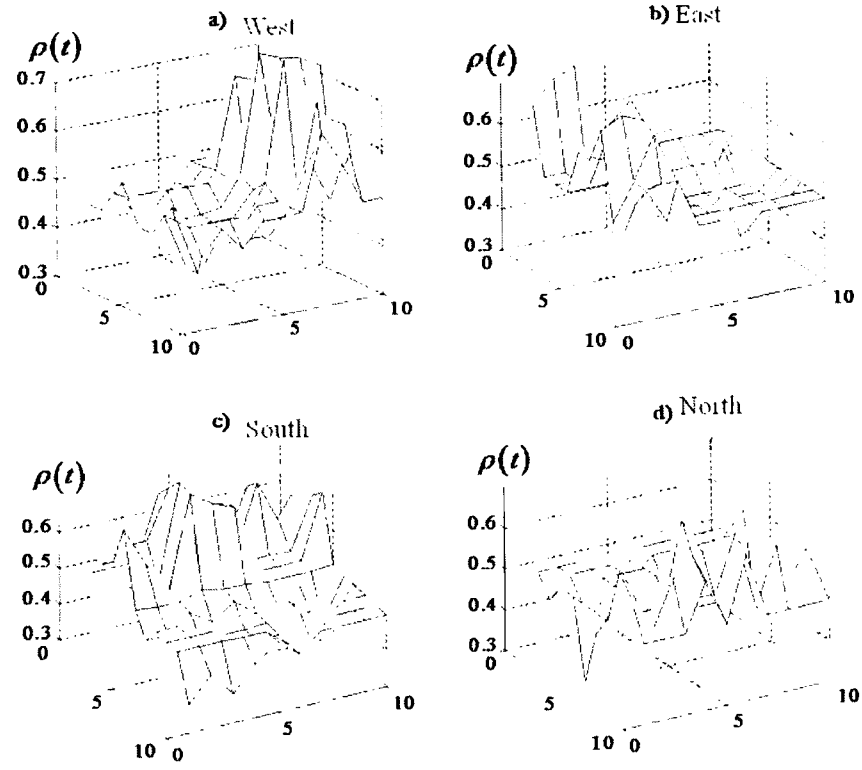

Figure 6.9: Random starting point GPFRL probabilities distribution.

Figure 6.8 Shows: a) the utility value distribution for SARSA algorithm and b), the utility value distribution for the Q-Learning algorithm; obtained after 100 trials of the random walk experiment, starting from a random position towards a static goal.

Figure 6.9 shows the probabilities of success of taking direction a) West, b) East, c) South, and d) North; obtained after 100 trials of the random walk experiment using the GPFRL algorithm, starting from a random position towards a static goal.

6.2.1.2.4 Discussion

The fixed start experiment is an example of a completely deterministic environment with no stochasticity in the inputs or outputs (i.e. the stochastic signal generator is set to $\varepsilon = 0.01$). For this experiment, the relative position between the start and the goal states are the same for all the trials. It can be observed that the proposed algorithm exhibits a faster convergence into the optimum path, whilst SARSA and Q-Learning learned the correct path in similar time but in a slighter slower time. This is attributed to the way the temporal difference error updates the value function in every time step as a in the SARSA and Q-Learning the learning agent must wait to reach the goal in order to update the value function.

It is also important to notice that the optimal value of α for SARSA and Q-Learning is the same (0.4), whilst for the GPFRL is significantly lower (0.003); since there is no stochastic signal added to the system, only one value function update is necessary for the learning agent to differentiate states and take an adequate decision for every state; so after the first trial the learner have coarse idea of the optimal path. In the case of SARSA and Q-Learning, the learner, doesn't have any information about the optimal path until it reaches a state where the value function has been previously updated. It becomes obvious that the addition of a stochastic signal will "confuse" in the first trials, as any initial coarse

information it have becomes modified randomly forcing the learner to go through the same state several more times in order to be able to differentiate the decisions to make for every state.

6.2.1.3 Stochastic Environment

6.2.1.3.1 System description

This section presents results on a different version of the grid world called the windy grid world problem with the addition of stochasticity. The purpose of this experiment is to evaluate the influence of environment stochasticity on the performance of our proposed GPFRL and compare it with that of SARSA and Q-Learning. This is done by adding two kinds of stochasticity: policy stochasticity and environment stochasticity into the windy grid world, verifying the advantages of the GPFRL in a broader setting.

Uncertainties in the input states become inherent under these stochastic conditions, allowing us to evaluate the uncertainty handling ability of our proposed method.

As in the previous case, the agent has to move from the start (S) to goal (G) state. In the windy grid world version there is a crosswind upward through the middle of the grid in such a way that the final cell is shifted upward by stochastic wind. The strength of which varies from column to column. The numbers under the grid represents the wind strength in the column above, as seen in Figure 6.10, and indicate the number of cells shifted upward. For example, if the agent is one cell to the right of the goal, then the action left takes it to the cell just above the goal. Let us treat this as an undiscounted episodic task, with constant rewards of -1 until the goal state is reached.

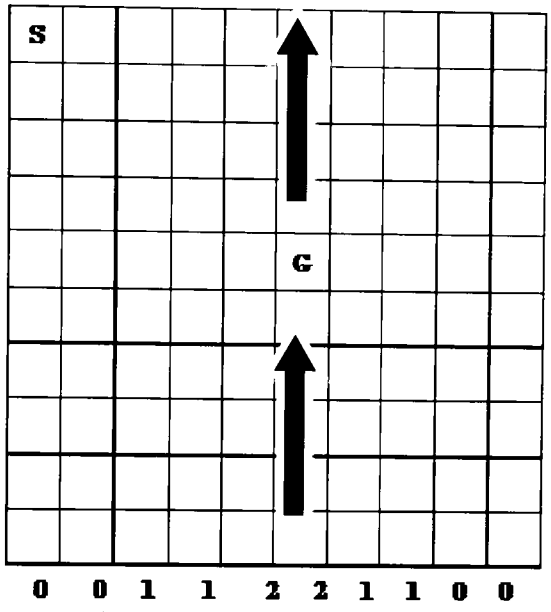

Figure 6.10: Windy grid world.

In Figure 6.10 the arrows shows the direction of the stochastic wind, so that the number under the columns represent the number of cells the learning agent will be shifted in direction of the wind when it reaches its corresponding column.

We also added environment and policy stochasticity to the windy grid world problem and compared the performance results with the deterministic case in order to evaluate the uncertainty effect over the learning agent. The environment stochasticity was added by moving the agent with a probability of 20% in a random direction instead of the direction corresponding to the action. The policy stochasticity was added by using ε=0.2 instead of 0.01.

6.2.1.3.2 Windy grid world with stochastic signal

Table 6.2 shows the used parameters for SARSA algorithm and the GPFRL algorithm under the windy grid world experiment.

TABLE 6.2 PARAMETERS USED FOR THE RANDOM WALK EXPERIMENT.

Parameter	SARSA	GPFRL

α	0.1	0.00025
β	-	0.005
γ	0.95	0.95
ε	0.2	0.2

For this test, the system was set to execute 40 trials and the results were stored and averaged for 100 plays. Figure 6.2 shows the average number of steps the agent takes for reaching the goal state from a static starting position in the grid world. It can be observed that the system not only converges to an optimum faster than the SARSA algorithm, but also that it does so with a considerably lower steps in the first trial a quick convergence to zero and solid stability thereafter.

It is to note, that in our experiments the Q-Learning algorithm never converged. The addition of a policy other than a greedy one produced a divergence in the Q-values found by the algorithm. The present results will only show a comparison between SARSA and the proposed GPFRL.

Figure 6.11 shows the results of the windy grid world experiment for SARSA and the GPFRL algorithm. The GPFRL algorithm shows handled the added stochasticity and learned the optimal (shortest) path to the goal in a short time.

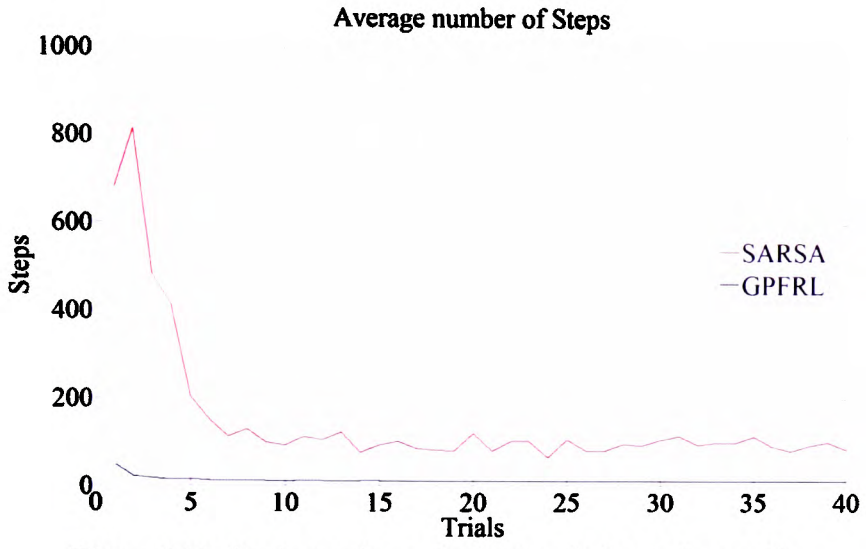

Figure 6.11: Static starting point in the windy random walk.

The standard deviation values for the data in Figure 6.11 are shown in Figure 6.12. It can be observed here that after the GPFRL have found the optimal path it follows it ever after; this can be seen in the standard deviation values that approximate to 0. On the other side, SARSA, not only does not converge to the optimal path but it succeeds to find the goal states in a large number of steps with a standard variation of around 100.

Figure 6.13 shows the distribution of the Q-values after a learning trial for the SARSA algorithm under the windy grid world experiment. The large flat area is understood as an un-explored area which never leads the agent to the goal state.

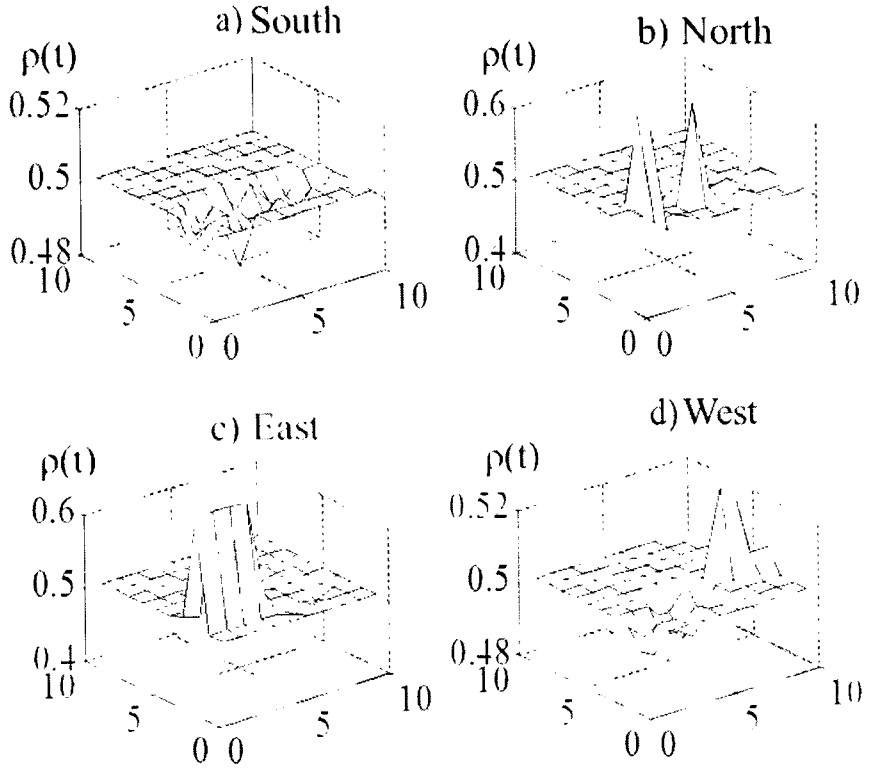

Figure 6.14 shows the probabilities of success of taking direction a) West, b) East, c) South, and d) North; obtained after 100 trials of the random walk experiment using the GPFRL algorithm, starting from a random position towards a static goal.

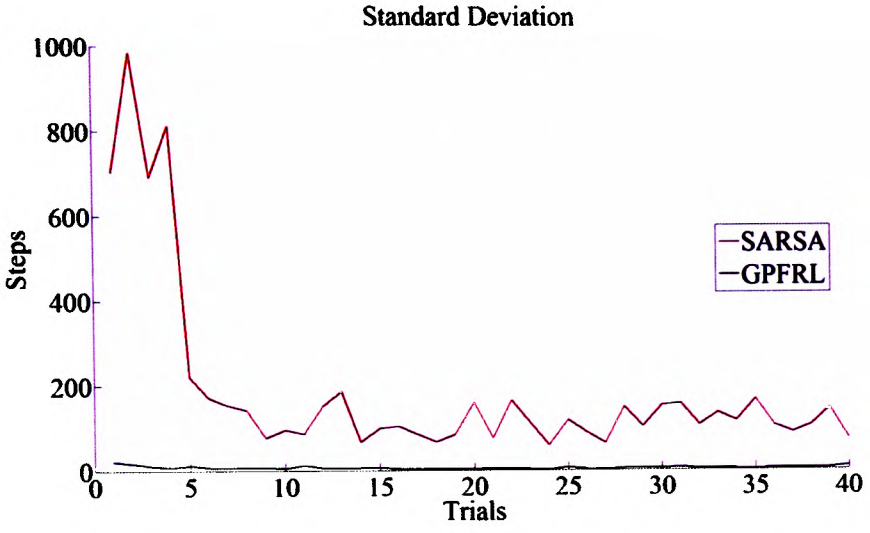

Figure 6.12: Static starting point in the windy random walk.

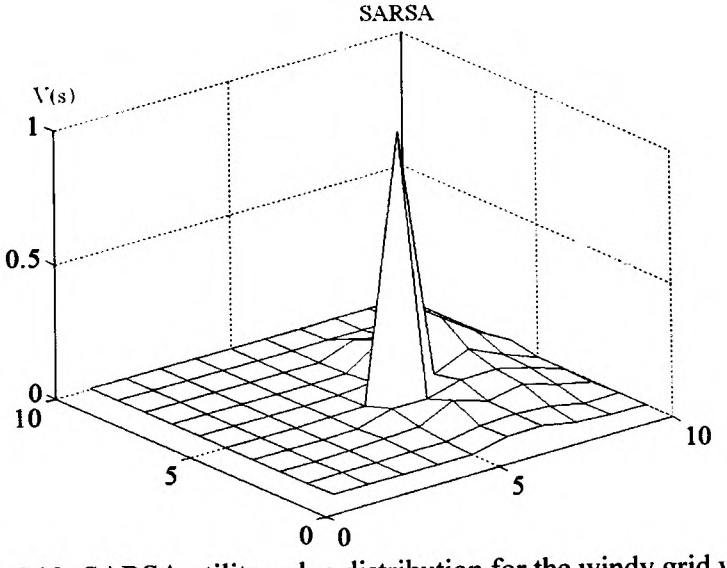

Figure 6.13: SARSA utility value distribution for the windy grid world.

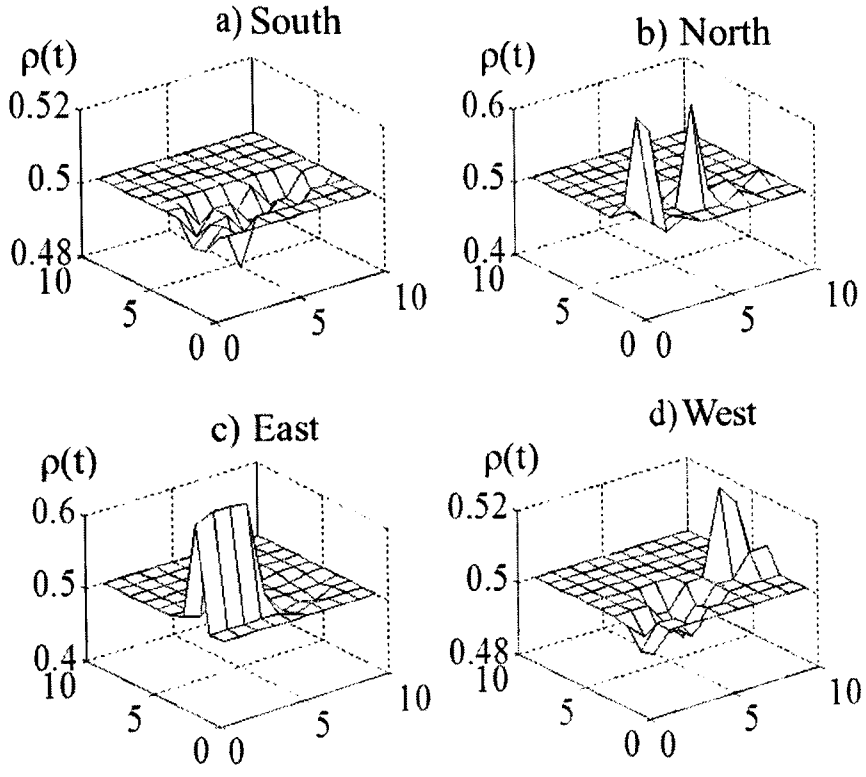

Figure 6.14: GPFRL probabilities distribution for the windy grid world.

6.2.1.3.3 Discussion

In this experiment an important observation is the reduction of the optimal update rate, α in contrast with the previously studied in the experiment proposed in chapter 6.2.1.2. As stochasticity increases, a high update rate does not directly contribute the learning speed as the information collected from the environment is not reliable (due to the added uncertainty) it can be better understood if a small value of α is considered as a cautious decision, in which case a larger number of decisions are needed in order for the system to find the correct decisions.

With the added stochasticity, the Q-Learning method failed in all trials to learn the optimal solution thus, its results were removed from the c omparison graphs.

Whilst SARSA was able to find a path between the start state and the goal state, its solution was not optimal in all cases. As a contrast, the proposed GPFRL learned the path between the start and the goal state in a few trials, and its solution was optimal in all cases, thus proving the uncertainty robustness of the proposed method.

6.2.1.4 Conclusions

The GPFRL is conceptually simple which relies on well-established ideas from Bayesian statistics. A decisive difference between the proposed GPFRL and common RL algorithms, as SARSA and Q-Learning, is the use of a probabilistic model for the transition dynamics, which mimics two important features of biological learners: the ability to generalize and the explicit incorporation of uncertainty into the decision making process. These model uncertainties have to be taken into account during long-term planning to reduce model bias. Beyond specifying a reasonable cost function, GPFRL does not require expert knowledge to learn the task.

In the presented experiment, the GPFRL algorithm was tested in a grid world environment under different conditions. First a standard deterministic version of the grid world was used to test the SARSA, Q-learning and GPFRL algorithm in a random walk using a fixed starting state and following a random starting state. The purpose of this experiment was to test the ability of our algorithm to handle randomness and to test its exploration/exploitation characteristics. The results can be observed in Figure 6.2 and Figure 6.6 for the fixed starting state and the random starting state respectively. The obtained results were conclusive; the GPFRL outperformed the classic SARSA and Q-learning algorithms in terms of learning rate. The GPFRL converged to the optimal solution in a shorter time and showed good stability in both experiments. In the random starting state experiment the difference was more remarkable. The GPFRL algorithm found the

optimal (shortest) path in fewer than ten trials, whilst SARSA and Q-learning algorithms required around fifty trials with very unstable initial choices.

In the second set of experiments, the GPFRL was tested in a highly stochastic version of the grid world, the windy grid world. In this case, "wind" was added to the standard grid world, in order to add environment stochasticity. Also the greedy term was increased from 0.01 (greedy behaviour) to 0.2 (more exploration) in order to add randomness. Under these conditions Q-learning failed to converge whilst SARSA converged to a non-optimal policy. Again the GPFRL showed a strong uncertainty resistance, by converging to the optimal policy in three trials and showed a very stable behaviour thereafter.

In all the experiments described, an important factor that contributed to the fast learning speed is the use of a richer reward scheme, where an internal reinforcement signal is provided. Whilst in SARSA and Q-Learning the agent receives a positive reward every time the agent reaches the goal state, in the GPFRL method, the agent receives a reward in every state. This reward is calculated by shaping the external reward (given at the goal state) by using temporal difference as described in (5.10). This internal reinforcement or reward signal represents an estimation of the reward for the current state as a difference between the prediction of eventual reinforcement of the current state and that of the previous state, as it was first described by (Barto et al., 1983).

Optimal design of reward functions has been studied before as in (Laud and DeJong, 2003, Mataric, 1994) where different experiments showed faster learning rates. In 1998, (Dorigo and Colombetti, 1998), suggested the use of reward shaping which import behaviourist concepts and methodology into RL, and discussed a model for automatic training of a RL agent. In the scheme they consider, the automatic trainer has an abstracted specification of the task, and it automatically rewards the agent whenever its behaviour is a better approximation

of the desired behaviour. However, it has been suggested (Marthi, 2007) that this notion of shaping goes well beyond of using a rich reward scheme.

It is clear that a well-designed reward function may facilitate learning, promote faster convergence, and prevent aimless wandering. Also, as pointed out by Laud (Laud and DeJong, 2003), if the optimal value can be provided as a reward, the RL task successfully collapses to greedy action selection.

Beyond this specific application, however, the larger and more important issue is whether learning from experience can be useful and practical for more general complex problems. Certainly the quality of results obtained in this study suggests that the approach may work well in practice, and may work better than we have a right to expect theoretically

6.2.2 Mobile Robot Obstacle Avoidance

The two most important topics in mobile robot design are planning and control. Both of them can be considered a utility optimization problem, in which a robot seeks to maximize the expected utility (performance) under uncertainty (Thrun, 2000).

In order to increase the flexibility of robots, facing unforeseen changes such as changes in their environments or sensor failure, the amount of predefined knowledge used in the control strategy has to be kept as low as possible. The present experiment, presents an automatic learning algorithm that uses reinforcement learning in order to find sensor-motor couplings through the robot's interaction with the environment.

Although the results obtained in many different test and simulated domains look promising, RL techniques have rarely been implemented in application requiring

real robots. Robotic applications present difficult challenges to RL methods, nevertheless its experience based motivation, high reactivity and effectively layered structure are still of great potential.

While recent techniques have been successfully applied to the problem of robot control under uncertainty (La, 2003, Pineau et al., 2003, Poupart and Boutilier, 2004, Roy et al., 2005), they typically assume a known (and stationary) model of the environment. This dissertation, discusses the problem of finding an optimal policy for controlling a robot in a partially observable domain, where the model is not perfectly known, and may change over time; whilst proposing that a probabilistic approach is a strong solution not only to the navigation problem, but also to a large range of robot problems that involves sensing and interacting with the real world. However, few control algorithms make use of full probabilistic solutions and as a consequence; robot control can become increasingly fragile as the system's perceptual and state uncertainty, increase.

Reinforcement learning enables an autonomous mobile robot to sense and act in its environment to select the optimal actions based on its self-learning mechanism. Two credit assignment problems should be addressed at the same time in reinforcement learning algorithms, i.e., structural and temporal assignment problems. The autonomous mobile robot should explore various combinations of state-action patterns to resolve these problems.

In our third experiment, the GPFRL algorithm was implemented on a Khepera III mobile robot (Figure 6.15) in order to learn a mapping that will enable itself to avoid obstacles. For this case, four active learning agents were implemented to read information from infrared (IR) distance sensors and learn the probabilities of success for the preselected actions.

Figure 6.15: Khepera III mobile robot.

In this experiment the GPFRL algorithm is implemented in a personal computer with a wireless link to a Khepera III mobile robot. All the computations were performed offline and the data was send and received via a Bluetooth link between the PC and the robot. The software was developed in C++ using Microsoft Visual Studio.

The algorithm consists in two software threats, were one thread manages the communication and the second executes the GPFRL algorithm.

Thread one, requests data from the robot, the robot receives the request, and sends the data corresponding to the values of its IR sensors, the PC sends a speed command for each wheel of the robot and receives a robot acknowledge of the successful transmission, then the whole process repeats. All the information is stored in a blackboard, where the speeds of the wheels and the IR data is continuously written and updated.

The second thread reads the IR data from the blackboard, and selects an appropriate action using a probabilistic fuzzy logic controller where the rules are updated every cycle using the proposed GPFRL algorithm. The selected action is then transformed into wheel speeds and the information is written in the blackboard.

6.2.2.1 The Input Stage

The Khepera III robot is equipped with 9 IR sensors distributed around it. For every time-step an observation $o_t \in O$ of the system state is taken. It consists of a sensor reading $x_k(t)$, k = 1, 2,..., l where l is the number of sensors and x_k is the measured distance at sensor k. The value of x_k is inversely proportional to the distance between the sensor and the obstacles where $x_k(t)=1$ represents a *distance=0* between sensor k and the obstacle and $x_k(t)=0$ corresponds to an infinite distance.

The input stage is composed by two layers: the input layer and the regions layer as shown in Figure 6.17.

In order to reduce the dimensionality problem a clustering approach was used. Signals from sensors 2, 3, 4, 5, 6 and 7 are averaged (weighted average using the angle with the orthogonal as a weight). They inform about obstacles in the front. Signals from sensors 1, 8 and 9 are grouped together and averaged. They inform about obstacles behind the robot as it can be observed in Figure 6.19. Another clustering procedure is applied to signals coming from sensors 1, 2, 3 and 4, which inform about obstacles to the right, and IR sensors 5, 6, 7 and 8, which inform about obstacles to the left.

$$x_k^R = \frac{\sum_{s=s_1}^{s}\left(90-\theta_s^R\right)\times IR_s}{\sum_{s=s_1}^{s}\left(90-\theta_s^R\right)} \tag{6.3}$$

In (6.3) x_k^R the result of the averaging function, where R, $R=\{1,2,3,4\}$, represents the evaluated region: front, back, left, and right, IR_s is the input value

of sensor s and θ_s^R is the angle between the sensor orientation and the orthogonal line R.

Internally, the Khepera III robot receives a numerical value from the IR sensors. Figure 6.16 shows the relationship between the actual distance and the received value of the IR sensors of the Khepera III mobile robot. In this graph it can be observed the uncertainty in the measurements as a variation between an estimated range model and the raw sensor measurements, the graph in Figure 6.16 also shows the variation range of such measurements. The received value is inversely proportional with the distance and ranges between 0 and 4000 for distances between 0 and 25cm.

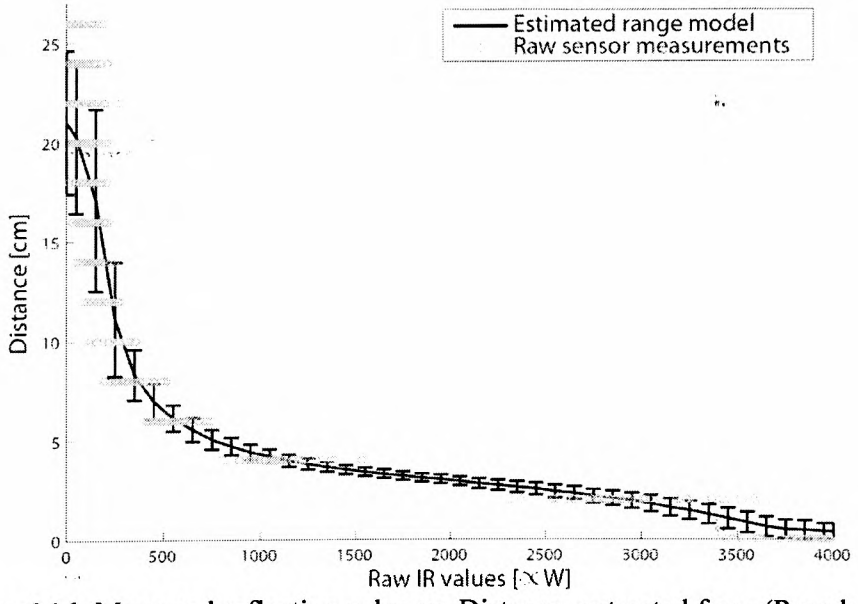

Figure 6.16: Measured reflection value vs. Distance, extracted from (Prorok et al., 2010).

Figure 6.17 shows how the nine signals acquired from the sensors are grouped inside the network into the 4 predefined regions. In this figure lines L1 and L2, are two of the four orthogonal lines.

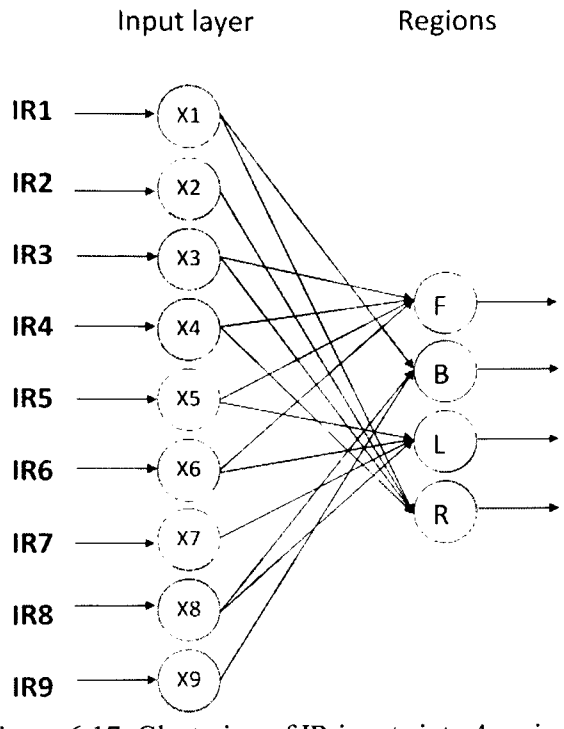

Figure 6.17: Clustering of IR inputs into 4 regions.

The signals from the averaging process are then fuzzified using Gaussian membership functions as specified in (6.4). Two membership functions (a left shouldered and a right shouldered) with linguistic names of "far" and "near" were used to evaluate each input in order to keep computation time to a minimum so that many sets of data could be analysed every second (approximately one every 30 milliseconds). At first we have considered the 3 membership functions: $\mu_k^L(t)$ -far distance, $\mu_k^Z(t)$ medium distance and $\mu_k^R(t)$ -near distance for each infrared sensor. Preliminary experiments with 3 membership functions combined into a number of 18 rules for the inference system which demanded a long time for calculations with no observable improvement in performance. In fact it had a negative impact on the learning, by decreasing its learning speed. For this reason we have decide to take only the following membership functions: $\mu_k^L(t)$ –far and

$\mu_k^R(t)$ –near, defined in eq. (6.4). These membership functions are depicted in Figure 6.18. The near and far values span over a normalization of the measured reflection value, where according to Figure 6.16 the crossing value correspond approximately to 4cm of distance between the sensor and the obstacle (wall).

$$\text{Far:}\quad \mu_k^L(t)=\begin{cases}1 & \text{if } x_k \le c^L\\ e^{-\frac{1}{2}\left(\frac{x_k(t)-c^L}{\sigma^L}\right)^2} & \textit{otherwise}\end{cases}$$

$$\text{Near:}\quad \mu_k^R(t)=\begin{cases}e^{-\frac{1}{2}\left(\frac{x_k(t)-c^R}{\sigma^R}\right)^2} & \text{if } x_k \le c^R\\ 1 & \textit{otherwise}\end{cases} \qquad (6.4)$$

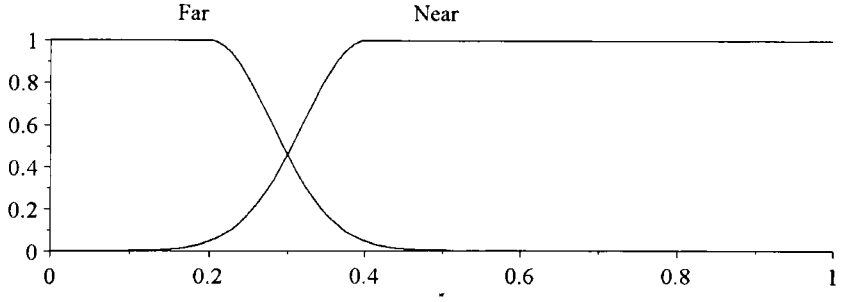

Figure 6.18: Membership functions for the averaged inputs

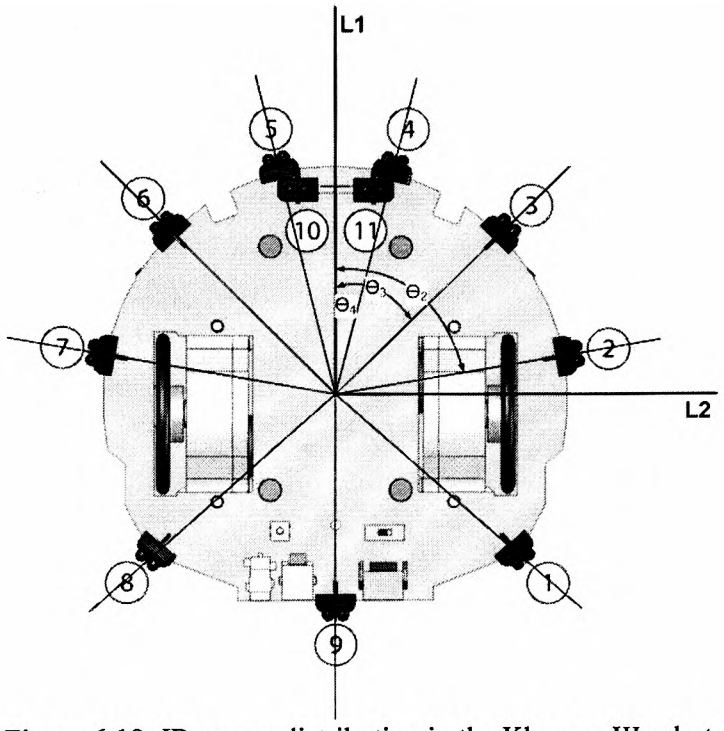

Figure 6.19: IR sensor distribution in the Khepera III robot.

The centre and standard deviation values used in Figure 6.18 were manually adjusted for this particular case and their values are shown in Table 6.3. For this experiment, using (6.3), clusters with average distances of 0.15 or below generate a negative reinforcement signal (reward) of -1 (robot collides with an obstacle), and zero reward is given in every other situation. We use a discount factor, γ of 0.95 which causes the learning system to prefer shorter solutions.

TABLE 6.3 CENTRE AND STANDARD DEVIATION FOR ALL MEMBERSHIP FUNCTIONS.

c^L	c^R	σ^L	σ^R
0.2	0.4	0.08	0.08

6.2.2.2 The Rules Stage

In this layer the number of nodes is the result of the combination of the number of membership functions that each region has and it is divided in two sub groups, each one of them triggers two antagonistic actions and are tuned by independent learning agents. Figure 6.20 shows the interconnection between the rules layer and the rest of the network.

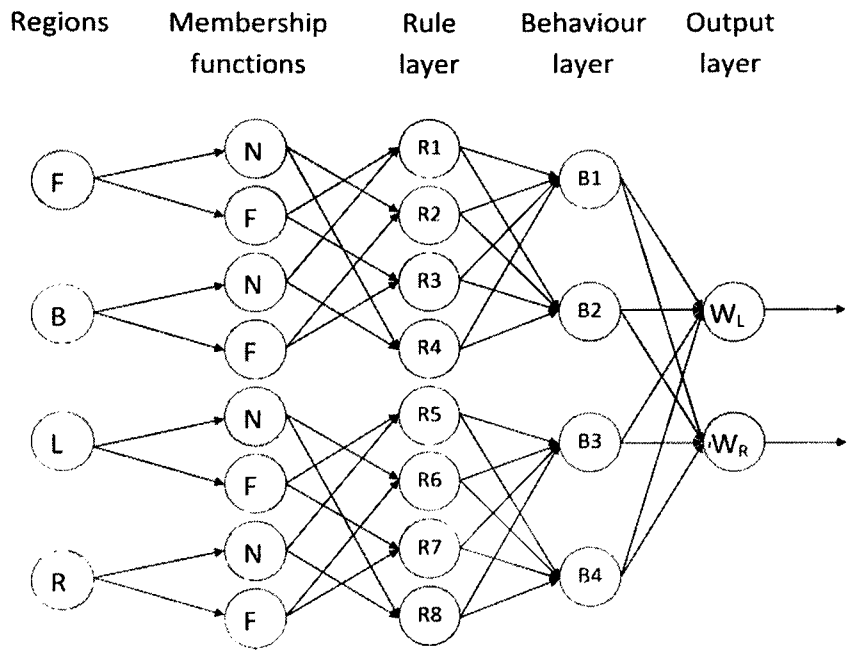

Figure 6.20: The controller structure.

As a result of the clustering method used, the number of rules is reduced from 16 rules to 8 rules. These groups of rules are independent from each other, and combine the information from different regions of the robot. Preliminary experiments using three and four membership functions in the inputs, showed no considerable increase in performance but a considerable delay in the learning time as the number of rules increased exponentially. Furthermore, a small number of rules make possible the addition of some *a priori* knowledge about avoiding obstacles, further reducing the learning time.

The result of the application of each rule is a preselected behaviour. For example, for rule R1:

If region F=Far and region B=Near then do B1 with $\rho = \rho_1$ *and B2 with* $\rho = \rho_2$

For each rule all the behaviours are selected but with different probability of success, then they are combined and defuzzified into wheel speed. The different behaviours or actions are described in the next section.

6.2.2.3 The Output Stage

Finally the output stage consists in an action selection algorithm. The action is described as a vector, $\vec{A}_j$, j = 1, 2,...,m, where "m" is the maximum number of actions, in this case, four, being:

- A_1 = Forward.
- A_2 = Backwards.
- A_3 = Turn left.
- A_4 = Turn right.

These actions are expressed as a vector where each term represents a relative wheel motion such that: forward: (1, 1), backwards: (-1, -1), turn right: (1, -1) and turn left: (-1, 1). The velocity is then computed using

$$\vec{v} = \sum_{j=1}^{m} \vec{A}_j \times P_j \tag{6.5}$$

where $\vec{v} = (v_1, v_2)$ and each of its terms represent the normalized speed of each of the wheels. Evaluating (6.5) on both maximum and minimum cases of P_j, obtain

$$\vec{v}_{max} = (2,2) \tag{6.6}$$

In order not to saturate the speed controller of the robot's wheels, the velocity for each wheel is expressed as:

$$V_{L/R} = \frac{1}{2} V_{max} \times v_{1/2} \tag{6.7}$$

In (6.7) V_{max} is the maximum allowed speed for the wheels of the robot and $V_{L/R}$ is the final speed commanded to each wheel.

6.2.2.4 Results

Figure 6.21 depicts the distance information coming from the IR sensors. The signal spikes indicate obstacle detection (wall). A flat top in the spikes indicates saturation was produced due to a crash against an obstacle. After the learning process, the robot avoids all the obstacles successfully. In our simulations based on real time, the average learning time was around 350 time steps corresponding to 7 seconds.

Table 6.4 shows the RL parameters used for this particular experiment.

TABLE 6.4 COEFFICIENT VALUES FOR THE RL ALGORITHM.

α	β	γ
7	0.03	0.95

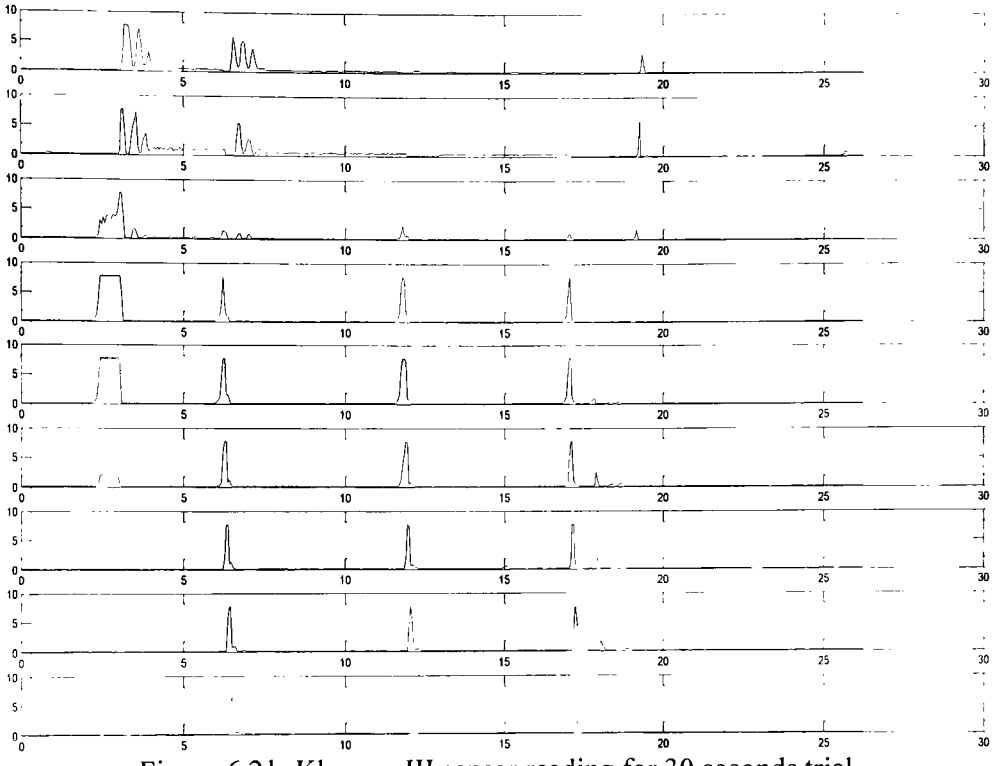

Figure 6.21: Khepera III sensor reading for 30 seconds trial.

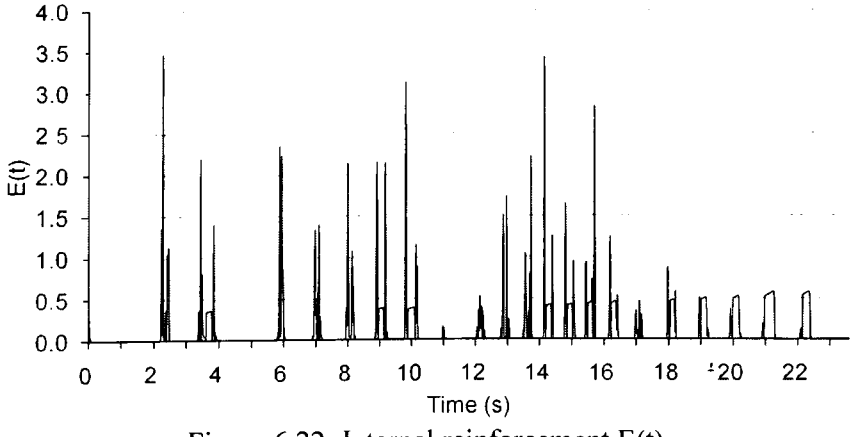

Figure 6.22: Internal reinforcement E(t).

Figure 6.22 represents the squared average of the reinforcement signal versus time, ach one of the spikes represents high reinforcement values (in this case negative reinforcements) which were given at every collision, after around 16

seconds the spikes are reduced (truncated) showing that the Khepera III mobile robot has effectively learned to avoid obstacles.

6.2.2.5 Conclusions

The proposed GPFRL is more practical than the basic fuzzy controller since the FIS is self-tuning based on the reinforcement signal. It is more practical to obtain reinforcement signal that only give rewards and punishments with respect to the states reached by the robot rather than desired behaviours of all situation in which the robot has to do.

In this experiment the GPFRL algorithm was has been exploited to facilitate real-time dynamic learning and control of a Khepera III mobile robot. The GPFRL generalizes the continuous input space with fuzzy rules and has the capability of responding to smoothly varying states with smoothly varying actions using fuzzy reasoning. Additionally it is possible for prior knowledge to be embedded into the fuzzy rules, enabling the robot to explore interesting environments and reducing the training time significantly.

In preliminary experiments, where the learning agents in the Khepera III robot were configured to use a greedy policy (the learning agent chooses the action with maximum desirability, which is considered satisfactory thereafter) resulted in sub optimal solutions (e.g. a backwards and forward motion, in case when a straightforward motion is optimal).

Some differences have been found in the Khepera's ability to negotiate different environments with the effectiveness of the avoidance learning system varying for different configurations of obstacles. However, only limited performance loss has been observed in transferring from a learned environment to a new one, which is quickly compensated if the Khepera is allowed to adapt its strategies to suit the

new circumstances. Therefore it can be concluded that the learning system is capturing some fairly general strategies for obstacle avoidance.

Since the primary task of obstacle avoidance is satisfied, a further improvement of optimality of solution should include a form of stochastic action exploration and a longer action sequence history. But these aspects go beyond the scope of this dissertation. Moreover, to achieve the global path planning navigation a goal seeking behaviour must be included and coordinated with the local obstacle avoidance task.

6.3 Control Experiments

6.3.1 Cart-Pole Balancing Problem

In order to assess the performance of the proposed approach, the GPFRL algorithm is implemented in a simulated cart-pole balancing system. This model was used to compare GPFRL (for both discrete and continuous actions) to the original AHC (Barto et al., 1983), and other related reinforcement learning methods.

For this case the membership functions (centres and standard deviations) and the actions are preselected. The task of the learning algorithm is to learn the probabilities of success of performing each action for every system state.

6.3.1.1 System Description

The pole-balancing problem is a pseudo-standard benchmark problem from the field of control theory and artificial neural networks for designing and testing controllers on complex and unstable nonlinear systems. The cart-pole system, depicted in Figure 6.23, is often used as an example of inherently unstable and

dynamic systems to demonstrate both modern and classic control techniques, as well as the learning control techniques of neural networks using supervised learning methods or reinforcement learning methods. In this problem a pole is attached to a cart which moves along one dimension. The control tasks is to train the GPFRL to determine the sequence of forces and magnitudes to apply to the cart in order to keep the pole vertically balanced and the cart within the track boundaries for as long as possible without failure. Four state variables are used to describe the system status, and one variable represents the force applied to the cart. These are: the displacement x and velocity $\dot{x}$ of the cart, and the angular displacement θ and its angular speed $\dot{\theta}$. The action is the force f to be applied to the cart. A failure occurs when $|\theta| \geq 12°$ or $|x| \geq 2.4m$. The success is when the pole stays within both these ranges for at least 500,000 time steps.

The differential equations for the dynamics of the cart-pole system are the ones proposed by (Barto et al., 1983). Equation (6.8) is the differential equation of motion of the pole; equation (6.9) is the differential equation of motion of the cart. The discrete time equations for the cart position and velocity are shown in (6.10) and the discrete time equations for the pole angle and angular velocity are shown in (6.11).

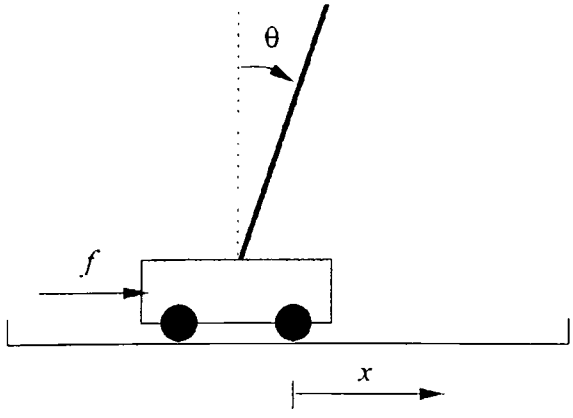

Figure 6.23: Cart-pole balancing system.

$$\ddot{\theta} = \frac{g\sin\theta + \cos\theta\left[\dfrac{-f - ml\dot{\theta}^2\sin\theta + \mu_c\,\mathrm{sgn}(\dot{x})}{m_c + m}\right] - \dfrac{\mu_p\dot{\theta}}{ml}}{l\left[\dfrac{4}{3} - \dfrac{m\cos^2\theta}{m_c + m}\right]} \tag{6.8}$$

$$\ddot{x} = \frac{f + ml\left[\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta\right] - \mu_c\,\mathrm{sgn}(\dot{x})}{m_c + m} \tag{6.9}$$

$$\begin{aligned} \theta(t+1) &= \theta(t) + \tau\cdot\dot{\theta}(t) \\ \dot{\theta}(t+1) &= \dot{\theta}(t) + \tau\cdot\ddot{\theta}(t) \end{aligned} \tag{6.10}$$

$$\begin{aligned} x(t+1) &= x(t) + \tau\cdot\dot{x}(t) \\ \dot{x}(t+1) &= \dot{x}(t) + \tau\cdot\ddot{x}(t) \end{aligned} \tag{6.11}$$

where g is the acceleration due to gravity, m_c is the mass of the cart, m is the mass of the pole, l is the half-pole length, μ_c is the coefficient of friction of cart on track, and μ_p is the coefficient of friction of pole on cart. The values used are the same as the ones used by (Barto et al., 1983) being:

TABLE 6.5 CART-POLE MODEL PARAMETERS.

Parameter	Description	Units	Value
θ	Angle of the pole in radians	rad	
$\dot{\theta}$	Angular velocity of the pole	rad/s	
$\ddot{\theta}$	Acceleration of the pole	rad/s2	
x	Cart position, measured as a relative offset from the middle of the track	m	
$\dot{x}$	Velocity of the cart	m/s	
$\ddot{x}$	Acceleration of the cart	m/s2	
t	Time	s	
τ	Discrete integration time step for the simulation	s	0.02
f	Magnitude of the force applied to the centre of the carts mass at time	N	[-10, 10]
g	Acceleration due to the gravity	m/s2	-9.8
mc	Mass of the cart	kg	1

mp	Mass of the pole	kg	0.1
l	Half-pole length	m	0.5
μc	Coefficient of friction of the cart on the track	-	0.0005
μp	Coefficient of friction of the pole on the cart	-	0.000002

In order to solve these equations there are several methods available as: Euler, Heun, Runge-Kutta (RK4) and others. In general, the RK4 method is the one used most commonly if a fixed-interval method is required. The programming is not too arduous, and the accuracy is substantially better than Euler's or Heun's methods, good enough for many applications. Nevertheless, it is important to consider that purpose of this experiment is not to provide an accurate description of the system response but rather to provide a comparison and testing platform, therefore the Euler method is chosen, as this is been used in most of the works presented for comparison in this section. These equations were simulated by the Euler method using a time step of $\tau = 20ms$ (50Hz).

For the simulation of this system it will be assumed that:

- The electrical system for response is instantaneous.
- The wheels of cart do not slip.
- The motor torque limit is not encountered.

The design of the fuzzy logic controller, specifically, the membership functions parameters consider the following criteria:

- If there is no knowledge about the system, identical membership functions whose domains cover the region of the input space evenly are chosen, and for every possible combination of the input fuzzy variables, a fuzzy rule is considered.

- Since the number of rules depends on the number of inputs and the number of membership functions within each input, the fuzzification layer only uses 2 membership functions, so that the resulting number of rules is 16 (2^4).

- The parameters of the fuzzy membership functions are based around similar and well documented experiments by: (Berenji and Khedkar, 1992) who used a reinforcement learning approach to tune a FIS in order to control an inverted pendulum of similar characteristics to the one described in this dissertation.

- The final values of the membership functions have been found by trial and error based on experiment criteria from (Tanaka and Wang, 2001)

Table 6.6 presents the fuzzy logic controller parameters for the four system inputs, x, $\dot{x}$, θ, and $\dot{\theta}$.

TABLE 6.6 CART-POLE MEMBERSHIP FUNCTION PARAMETERS.

Input	Centre N	Centre P	Standard deviation
x	-1.5	1.5	0.93
$\dot{x}$	-0.5	0.5	0.31
θ	-1.8	1.8	1.12
$\dot{\theta}$	-0.5	0.5	0.31

Table 6.7 shows the selected parameters for our experiments, where α is the actor learning rate, β is the critic learning rate, τ is the time step in seconds, and γ is the temporal difference discount factor. The learning rates were selected based on a sequence of experiments as shown in the next section, the time step is selected to be equal to the standard learning rate used in similar studies and the discount factor was selected based on the basic criteria that for values of γ close to zero, the system is almost only concerned with immediate consequences of its action. For values approaching one, future consequences become a more important factor in

determining optimal actions. In reinforcement learning the biggest concern is the long-term consequences of actions, so the selected value for γ is chosen to be 0.98.

TABLE 6.7 PARAMETERS USED FOR THE CART-POLE EXPERIMENT.

α	β	γ	τ
45.0	0.000002	0.98	0.02

6.3.1.2 Results

The Pole Balancing problem is used to compare our GPFRL approach to other reinforcement learning methods. These methods are the original AHC described in (Barto et al., 1983) and ten other reinforcement learning methods, namely, Anderson's method (Lin, 1995), Lee's method (Lee, 1991), Berenji's GARIC architecture (Berenji and Khedkar, 1992), Lin's RNN-FLCS method (Lin and Xu, 2006), the FALCON-RL method (Lin and Lee, 1993), Jouffe's FACL method (Jouffe, 1998), C. K. Lin's RLAFC method (Lin, 2003), Wang's FACRLN method (Wang et al., 2007), C.-T. Lin's RNFCN method (Lin, 1995) and Zarandi's GRLFC method (Zarandi et al., 2008). This detailed comparison is presented in Table 6.9, from which it can be seen that our GPFRL system required the smallest number of trials. All GPFRL experiments have been obtained with an FIS made of 16 rules (two Gaussian membership functions for every input). Fields where information was not available are marked as N/I and fields with information not applicable are marked as N/A.

For this experiment, 100 runs were performed and a run ended when a successful controller was found or a failure run occurred. A failure run is said to occur if no successful controller is found after 500 trials. The number of pole balance trials was measured for each run and their statistical results are shown in Figure 6.24.

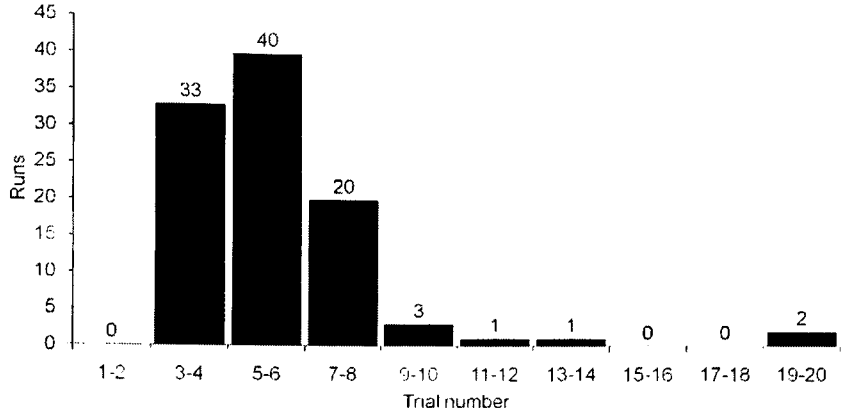

Figure 6.24: Trials distribution over 100 runs.

Table 6.8 shows the probabilities of success of applying a positive force to the cart for each system state after a complete learning run. It can be observed that values close to 50% are barely "visited" system states which can ultimately be excluded, furthermore reducing the number of fuzzy rules. This can also be controlled by manipulating the value of the stochastic noise generator, which can either force exploration, increasing the learning time, or it can force exploitation, which will ensure a fast convergence but less states will be visited.

TABLE 6.8 PROBABILITIES OF SUCCESS OF APPLYING A POSITIVE FORCE TO THE CART FOR EACH SYSTEM STATE.

$\dot{x}, x$	**N**			**P**		
	$\dot{\theta}, \theta$	N	P	$\dot{\theta}, \theta$	N	P
N	N	0.45	0.51	N	0.38	0.50
	P	0.77	1	P	0.57	0.71
	$\dot{\theta}, \theta$	N	P	$\dot{\theta}, \theta$	N	P
P	N	0.04	0.6	N	0.25	0.48
	P	0.56	0.61	P	0.53	0.53

Table 6.8 represents all the possible states for the cart-pole system which results from the combination of all the input membership values. The main division, groups the states according to the cart position and speed, and it is divided in four

sections. Each of these sections is further divided in four sections, according to the pole angle and angular speed. This generates a total number of possible states for the system of sixteen. For example, if we take the lower right corner of the first division of the table, this corresponds to a state where the position of the cart is positive and its speed is positive (moving to the right); logically this state will always require a force to be applied to the left (negative force) therefore, the probabilities of success of applying a force with a positive value, should be low. A further analysis including the position of the pole will assign lower probabilities for the case where the pole angle is negative and falling (negative radial speed), and higher probabilities for the case where the pole angle is positive and falling (positive radial speed) which correspond hierarchically to the values observed in Table 6.8.

After each iteration, the learning agent will try to use the current probability values as an input-output mapping; then a corresponding action will be executed, and a new reinforcement signal will be assigned according to the result of this action. After several iterations, the probabilities converge to their optimal values (e.g. the values shown in Table 6.8). This probability values are consider optimal, when two conditions are met:

1. Subsequent iterations produce little or no change to the probabilities.

2. The actions generated as a consequence of this probabilities, are able to balance the pole within the specified range, for the duration of a predetermined amount of time.

3. The actions generated as a consequence of this probabilities, are able to keep the cart within the specified range, for the duration of a predetermined amount of time.

The final optimal force to be applied to the cart every time step is calculated according to:

$$P_k(t) = \frac{\sum_{j=1}^{m} M_j(t) \cdot \rho_{jk}(t)}{\sum_{j=1}^{m} M_j(t)} \tag{6.12}$$

TABLE 6.9 LEARNING METHOD COMPARISON FOR THE CART-POLE BALANCING PROBLEM.

Learning method	Continuous states	Continuous actions	A priori knowledge	Number of fuzzy rules	Structure learning	Number of trials	Angular deviation (deg)	Uncertainty handling
AHC (Barto et al., 1983)	No	No	No	N A	No	75	4	No
Anderson (Lin, 1995)	Yes	No	No	N A	No	8000	N I	No
Lee's (Lee, 1991)	Yes	Yes	No	N A	Yes	8	N I	No
GARIC (Berenji and Khedkar, 1992)	Yes	Yes	Yes	13	No	300	1	No
RNN-FLCS (Lin and Xu, 2006)	Yes	Yes	Yes	35	Yes	10	1	No
FALCON-R1 (Lin and Lee, 1993, Wang et al., 2007)	Yes	Yes	Yes	10	Yes	15	0.5	No
FACL (Joufle, 1998)	Yes	Yes	Yes	54	Yes	3.7	N I	No
RLAFC (Lin, 2003)	Yes	Yes	Yes	81	No	1	N I	No
FACRLN (Wang et al., 2007)	Yes	Yes	No	11	Yes	8.68	1.5	No
RNFCN (Lin, 1995)	Yes	Yes	No	15	Yes	10	1	No
GRLFC (Zarandi et al., 2008)	Yes	Yes	Yes	23	No	3.35	N I	Yes*
GPFRL	Yes	Yes	Yes	16	No	3.33	0.4	Yes

The minimum and the maximum number of trials over these 100 runs are 2 and 10, with an average number of trials is 3.33 with only 4 runs failing to learn. It is also to be noted that there are 59 runs that took between 3 and 4 trials in order to find correctly learn the probability values of the GPFRL controller.

In order to select an appropriate value for the learning rate, a value that minimizes the number of trials required for learning but at the same time minimizes the number of no-learning runs (a run in which the system executed one hundred or more trials without success), a set of tests were performed and their results are depicted in Figure 6.25-Figure 6.28.

In Figure 6.25 and Figure 6.26 it can be observed that the value of alpha does not have a direct impact on the number of failed runs (which is in average 10%) as it does with the learning speed, with a value higher than forty five there is no significance increase on the learning speed. For these tests the program executed the learning algorithm twenty two times, each time consisting of one hundred runs, from which the average was taken.

In the following graphs the solid black line represents the actual results, whilst the dashed line is a second order polynomial trend line.

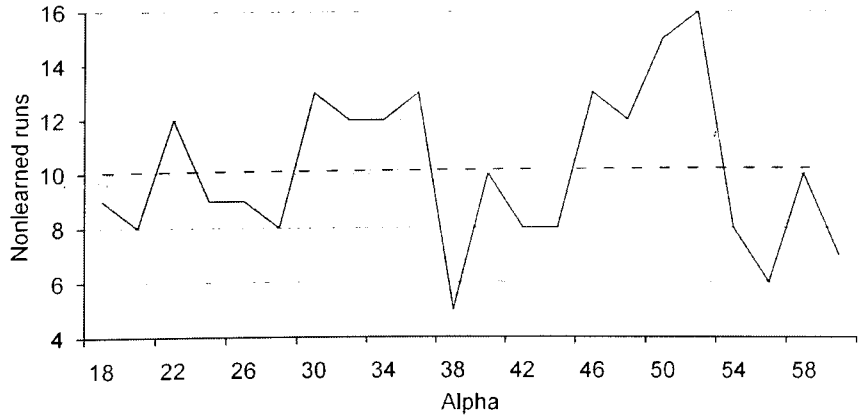

Figure 6.25: Actor learning rate, alpha, vs. number of failed runs.

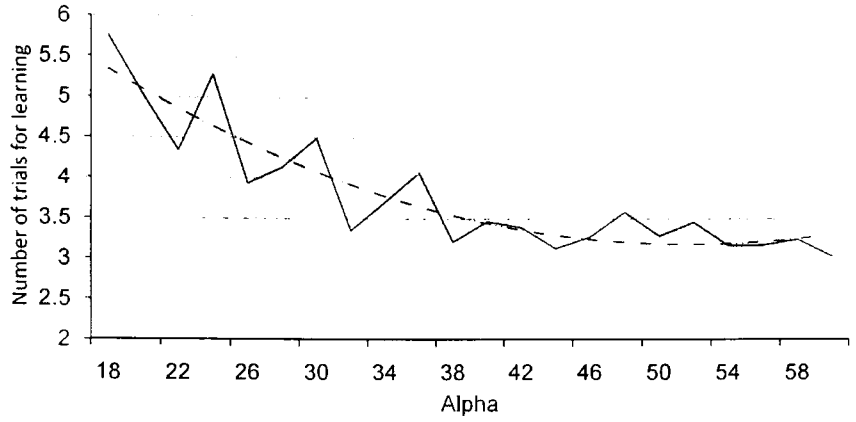

Figure 6.26: Actor learning rate, alpha, vs. number of trials for learning.

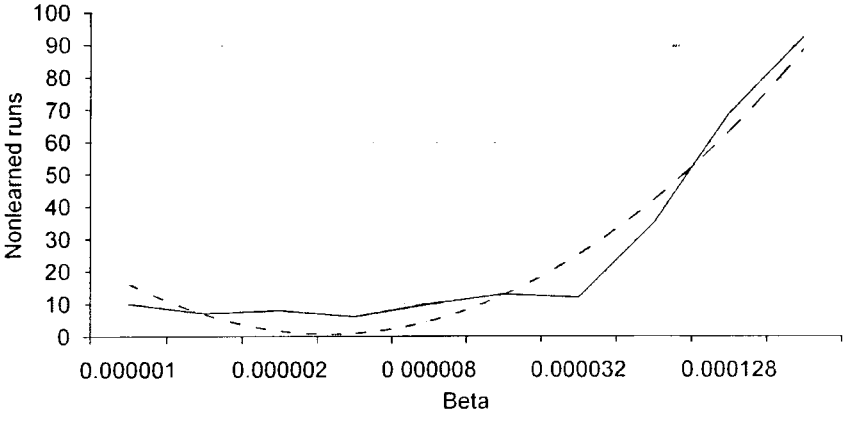

Figure 6.27: Critic learning rate, beta, vs. number of failed trials.

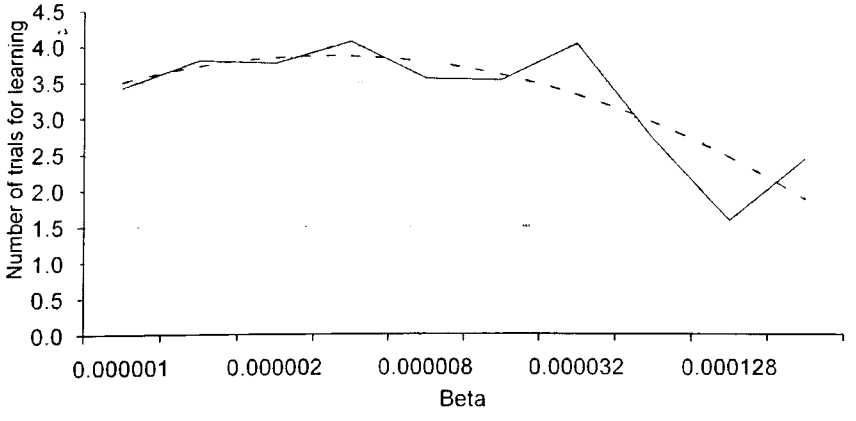

Figure 6.28: Critic learning rate, beta, vs. number of trials for learning.

For the second set of tests (Figure 6.27 and Figure 6.28), the values of beta were changed whilst keeping alpha at 45. It can be observed that there is no significant change in the number of failed runs with an increase of beta for values under 0.000032. With beta values over this, the number of non-learning runs increases quickly. Contrary, for values of beta below 0.000032 there is no major increase on the learning rate. The results of one of the successful runs are shown from Figure 6.29 to Figure 6.32.

Figure 6.29 shows the cart position with respect to the centre of the assigned area, the cart offset with respect with this centre has been observed to take different values in different experiments, with oscillations of similar values (close to a 0.6m peak to peak). It is an understandable effect, as the learner was not train to keep the cart near the centre; rather it was trained to keep the pole in balance and to keep the cart within certain limits.

The pole angle θ is shown in Figure 6.30, where oscillations centred on zero can be observed. As a difference with the cart, a successful pole balance can only be maintain if the pole is as vertical as possible, hence, oscillations of around 0.4deg, on average, peak-to-peak can be observed.

After a successful set of trials, the control force f is presented (Figure 6.31). It is important to notice that f is an external force and it is only applied to the cart in order to keep the pole balanced. The mean of the applied force is around 1 with a peak-to-peak value of ± 1.

The stochastic noise is shown in (Figure 6.32) and it was captured also at the end of a set of trials, consequently the display of small values, indicates that the learning agent has effectively explore many states, making the system to follow a greedy behaviour, where exploitation is preferred over exploration.

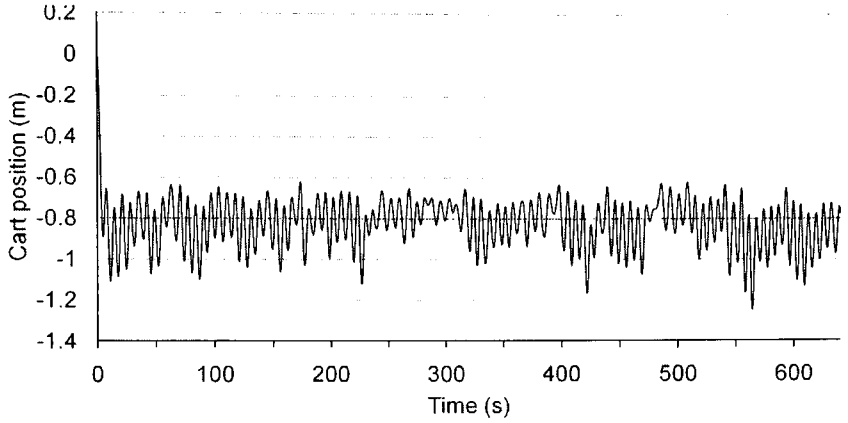

Figure 6.29: Cart position at the end of a successful learning run.

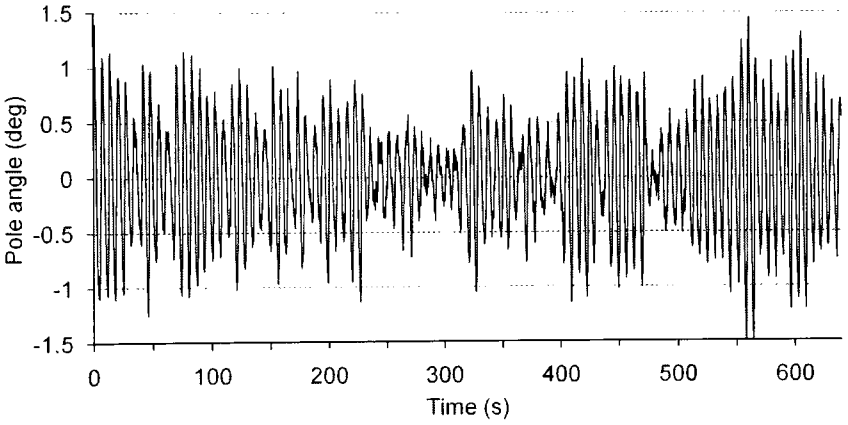

Figure 6.30: Pole angle at the end of a successful run.

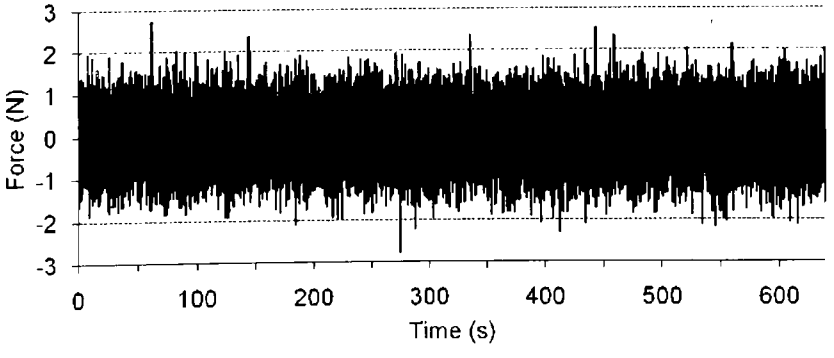

Figure 6.31: Applied force at the end of a successful learning run.

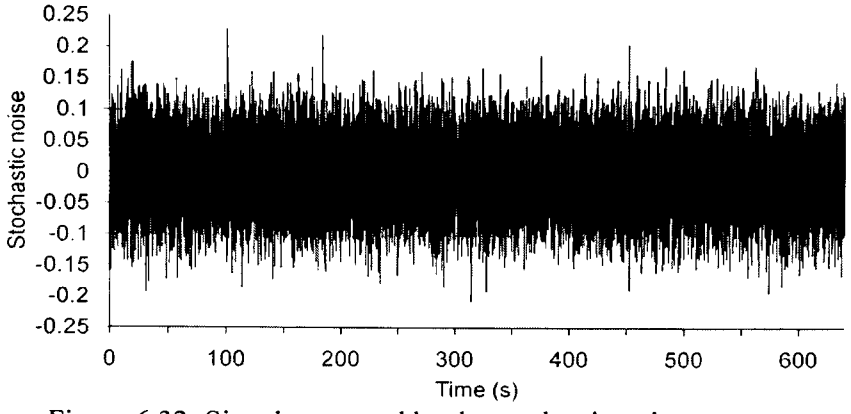

Figure 6.32: Signal generated by the stochastic noise generator.

6.3.1.3 Conclusions

In this dissertation a new method for optimizing fuzzy inference systems based on reinforcement learning and probabilistic theory has been proposed. In this section the proposed method was applied to a cart-pole system. The cart-pole system was controlled by a fuzzy inference system. The structure of the fuzzy inference system was fixed a priori.

Before learning, the average time until failure was about 25-100 time steps (up to 1 second). Within a few trials one could observe trials with well over 100000 time steps balancing time. After the learning, the fuzzy controller was able to stabilize the pendulum, initially and for comparison purposes, for 500,000 time steps. Further experimentation with the algorithm running unconstrained, the GPFRL algorithm was able to balance the pole for periods over 10million time steps (equivalent to 55 simulated hours).

The deficiencies in the input representation of the pole's state were overcome by using a fuzzyfier. The weak, initial strategy of random action-selection was transformed into a nearly deterministic choice of the better action for each state. Each action is quickly chosen by calculating the output of the controller (actor) every time step. These characteristics make this approach a well-suited option to

refining the control of a real-time task, and can do so even when given an inappropriate representation and with few clues as to what a good strategy would be.

An important thing to consider is that the proposed method was not configured to control the pole with a high precision; rather it was developed with the simple task of avoiding the pole to fall out of the specified angular range and avoid the cart to leave the specified distance range. Nevertheless the GPFRL balanced the pole with an angular deviation of 0.4deg from the vertical. This proposes the idea that perhaps redefining the goal of the learning, from learning to keep the pole within a range to learning to minimize the angular error between the pole and the vertical axis, could potentially reduce this angular deviation error, leading to more stable results, but with the cost of longer learning periods.

Finally, a number of alterations remain to be tried that will probably result in a faster learning. Some of the minor changes that can be made include:

- Adapting the parameter values during learning (optimization of membership function parameters and the number of rules) will potentially improve the performance of the proposed method.

- Using averages of past input values in updating weights rather than the single input from the previous step as done here.

- Also, a potentially great improvement in performance might result from the addition of a mechanism for learning a model of the problem even before goals are specified.

6.3.2 DC Motor Control

This experiment consists of a DC motor with a gear head reduction box. Attached to the output shaft there is a lever (considered to have no weight) of length L and at the end of this a weight W. The starting point (at which the angle of the output shaft is considered to be at angle 0) is when the lever is in vertical position, with the rotational axis (motor shaft) over the weight, so the motor shaft is exerting no torque. For this particular example the following parameters are used: L=0.5m and W=0.5kg.

Figure 6.33 shows the motor arrangement in its final position (reference of 90°)

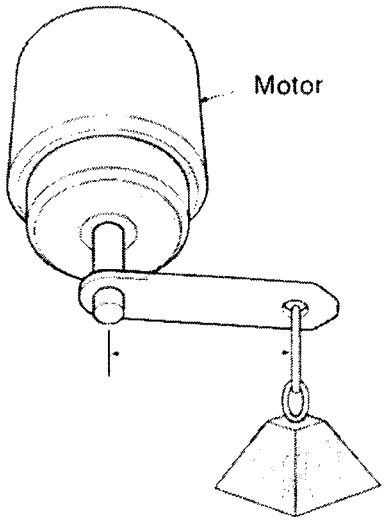

Figure 6.33: Motor with load attached.

The mathematical model used for this test is described by:

$$\dot{I} = \frac{V - (R_e \times I) - (K_e \times \dot{\theta})}{L} \tag{6.13}$$

$$\ddot{\theta} = \frac{(K_t \times I) - (D \times \dot{\theta}) - T}{J} \tag{6.14}$$

These equations can also be represented in state-space form. If we choose motor position, motor speed, and armature current as our state variables, we can write the equations as follows:

$$\frac{\partial}{\partial t}\begin{bmatrix} I \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} -R_e/L & -K_e/L \\ K_t/J & -D/J \end{bmatrix}\begin{bmatrix} I \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 1/L & 0 \\ 0 & -1/J \end{bmatrix}\begin{bmatrix} V \\ T \end{bmatrix} \tag{6.15}$$

In the above equations, I is the instant current in amperes, V is the instant voltage across the motor terminals, R_e is the winding resistance, K_e is the motor constant, $\dot{\theta}$ is the rotation speed of the rotor, K_t is the armature constant, D is the damping factor, T is the total torque applied to the motor rotor at a particular time step, and J is the moment of inertia generated by the rotor movement.

As with the car-pole example, it is important to consider that the purpose of this experiment is not to provide an accurate description of the system response but rather to provide a testing platform. Nevertheless, as the learning algorithm does not depend on the system model, the selection of a more accurate method will not have a considerable effect on the final performance, and will increase the design complexity unnecessarily. Finally, this motor model is solved using a time step τ of 0.00001 seconds, and are described:

$$\theta(t+1) = \theta(t) + \tau \cdot \dot{\theta}(t) \tag{6.16}$$

$$\dot{\theta}(t+1) = \dot{\theta}(t) + \tau \cdot \ddot{\theta}(t) \tag{6.17}$$

$$I(t+1) = I(t) + \tau \cdot \dot{I}(t) \tag{6.18}$$

The corresponding parameters used for this example are shown in Table 6.10:

TABLE 6.10 DC MOTOR MODEL PARAMETERS.

Parameter	Description	Unit	Value

Re	Electric resistance	Ω	0.611
Ke	Motor constant	V/rad/s	0.02587885
L	Electric inductance	H	0.00012
Kt	Armature constant	N.m/A	0.0259
D	Damping ratio	N.m/s	0.00000432
J	Moment of inertia of the rotor	(kg.m2/s2)	0.00000333

6.3.2.1 Control Signal Generation

For the present approach let's assume that there are only 2 possible actions to take: direct or reverse voltage, that is, the controller will apply either 24V or -24V to the motor spinning it clockwise or counter clock wise. At high commutation speeds the applied signal will have the form of a PWM whose duty cycle controls the direction and speed of rotation of the motor.

The selection of either of the actions described above will depend on the probability of success of the current state of the system. The goal of the system is to learn this probability through reinforcement learning.

The inputs to the controller are the error and the rate of change of the error as it is commonly used in fuzzy controllers.

6.3.2.2 Failure Detection

For the RL algorithm to perform, an adequate definition of failure is critical. Deciding when the system has failed and defining its bias is not always an easy task. It can be as simple as analysing the state of the system, like in the case of the classic cart pole problem (Barto et al., 1983); or it can get complicated if the agent needs to analyse an external object, for example in a welding machine, a positive reinforcement can be attributed if the weld has been successful or a failure if it has not.

For many modern control systems, the use of performance indices, constitute very important design tools available to Engineers. It provides a quantitative way to

measure a system performance in a mathematical way, so that an "optimum system" can be designed and evaluated. Four of the most well-known methods are: ISE, IAE, ITAE and ITSE which are defined as follows (Schapire, 1990):

$$IAE = \int_0^\infty |e(t)|\, dt \tag{6.19}$$

$$ITAE = \int_0^\infty t|e(t)|\, dt \tag{6.20}$$

$$ISE = \int_0^\infty e(t)^2\, dt \tag{6.21}$$

$$ITSE = \int_0^\infty t \cdot e(t)^2\, dt \tag{6.22}$$

Where $e(t)$ is the difference between the reference position and the actual motor position (system error). The IAE is computed at every time step, and is compared with a selectable threshold value. A control system failure is considered when the measured IAE is over the stated threshold thus generating a negative reinforcement and resetting the system to start a new cycle.

The proposed system continuously analyses the performance of the controller, through computing the integral of absolute error (IAE) and the (ITAE). These indexes were selected based upon experimentation, rather than a mathematical analysis. After some trial and error, the results obtained with the ITAE and IAE indexes consistently outperformed the results obtained with the ISE and ITSE indexes with respect to the final performance of the controller. In any case none of them proved to have a direct effect on the learning.

The IAE index is especially sensitive to system overshoot; changes in the reference can produce sudden changes in the error value making the IAE index less forgiving to overshoot. Introducing a time parameter into the IAE index, as in the I TAE, makes the index spe cially se nsitive to remaining errors over long periods of time, rather than larger errors produced on time steps near $t = 0$ this provides an excellent measure of steady state errors. By using these two performance indexes our proposed GPFRL system is able to configure the probabilistic controller to minimize both, overshoot and steady state errors.

In this example, the performance index criterion will penalize a response that has errors that persist for a long time, thus making the agent learn to reduce the overshoot and furthermore the steady state error.

Due to sudden changes in the reference signal used for this particular example the following method is used in evaluating the IAE and ITAE thresholds dynamically:

$$IAE_{\max}(t) = 1.8 + \int_0^{\infty} \frac{\left(\theta_{ref}(t) - \theta_{ref}(t-1)\right)^2 dt}{2000} \tag{6.23}$$

$$ITAE_{\max}(t) = \frac{IAE_{\max}(t)}{10} \tag{6.24}$$

Then, the reward signal "r" is defined according to (6.25)

$$r = \begin{cases} 0, \text{ if } ITAE < ITAE_{\max} \text{ and } IAE < IAE_{\max} \\ -1, \text{ if } ITAE \geq ITAE_{\max} \text{ and } IAE \geq IAE_{\max} \end{cases}, \nabla\, t < t_{\lim} \tag{6.25}$$

It immediately becomes obvious that for fixed reference signals over extended periods of time, the performance indexes will eventually overcome the maximum values; nevertheless this does not imply low performance of the controller. It is

imperative so, to establish a period of time over which the system IAE and ITAE should remain within acceptable limits ($t<t_{lim}$).

It is important to notice that the definition of an appropriate performance index and consequently, the corresponding reward signal, is highly dependent of the particular system under measure and its desired behaviour. For any given system, the definition of a rewarding or punishable circumstance can be more than a trivial problem with more than one solution. In the current example the selection of the IAE as a performance measure, can generate rewards that can "teach" the system to avoid overshoots. In similar way the selection of the ITAE as a performance measure, generates rewards (or punishments) that will lead the system to avoid steady state errors. It becomes clear that the appropriate selection of a performance measure can greatly affect the final performance of the controller, as each one tries to punish (or reward) different events.

6.3.2.3 System Configuration

The membership functions used to fuzzyfi the inputs are depicted in Figure 6.34 and Figure 6.35. Their parameters were selected based on experience, and are shown in Table 6.11 and Table 6.12.

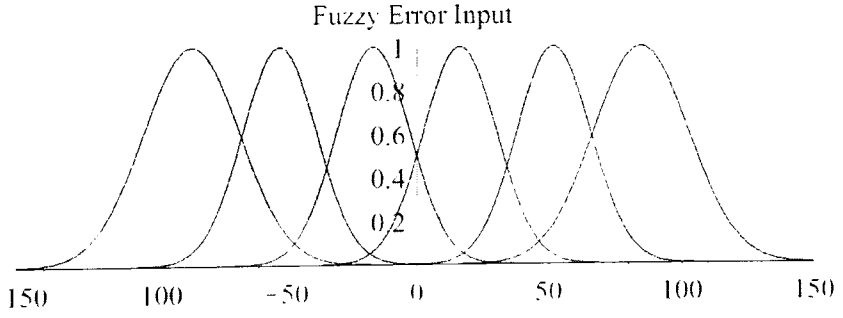

Figure 6.34: Membership functions of the error input.

TABLE 6.11 MEMBERSHIP FUNCTION PAREMETERS FOR ERROR INPUT.

	LargeN	MediumN	SmallN	SmallP	MediumP	LargeP
C	-85	-52	-16.5	16.5	52	85
σ	18	14	14	14	14	18

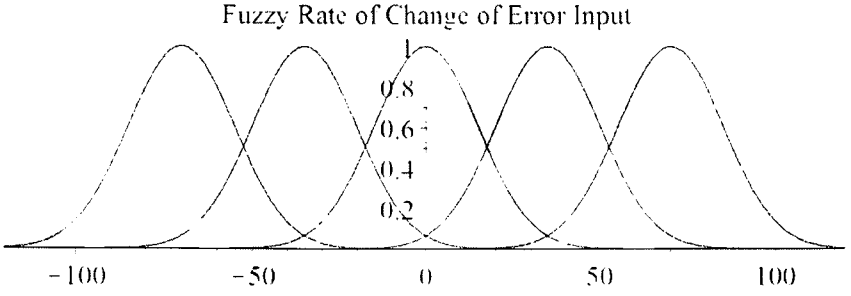

Figure 6.35: Membership functions for the rate of change of the error.

TABLE 6.12 MEMBERSHIP FUNCTION PARAMETERS FOR RATE OF ERROR INPUT.

	LargeN	SmallN	Centre	SmallP	LargeP
C	-70	-35	0	35	70
σ	15	15	15	15	15

The Reinforcement Learning agent requires some parameters to be defined. Table 6.13 shows the values selected by following two selection principles (Wang et al., 2007).

First, by adjusting the discount factor γ, we were able to control the extent to which the learning system is concerned with long-term versus short-term consequences of its own actions. In the limit, when $\gamma = 0$, the system is myopic in the sense that it is only concerned with immediate consequences of its action. As γ approaches 1, future costs become more important in determining optimal actions. Because we are rather concerned with long-term consequences of its actions, the discount factor had to be large, i.e., we set $\gamma = 0.95$.

TABLE 6.13 COEFFICIENT VALUES FOR THE RL ALGORITHM.

α	β	γ

7	1	0.95

Second, if a learning rate is small, the learning speed is slow. In contrast, if a learning rate is large, the learning speed is fast, but oscillation occurs easily. Therefore, we chose small learning rates to avoid oscillation, such as $\alpha = 7$, $\beta = 1$

6.3.2.4 Test Bench

A screen capture of the developed software is shown in Figure 6.36. The control window provide the choice of visualizing the response of the motor (angle) or the error; in the lower part it provides a real time visualization of the probabilities of success of applying a positive voltage to the motor (rotate the rotor clockwise).

In the upper part 4 different sections provide the following options:

- **RL Parameters:** Steps to balance is not used for this example, max errors, is the maximum number of failures allowed on each run, random range allows the user to indicate the standard deviation of the Gaussian random error generator in percentage, so that a 20% will correspond to a random number generated with a Gaussian distribution of mean zero and standard deviation of 0.4.

- **Control Parameters:** Reference angle indicates the angle the motor is expected to be (reference), the motor angle shows the current angle of the motor shaft, error, shows the difference between the current shaft angle and the reference angle, and rate of error, indicates the speed at which the error is changing.

- **Fuzzy Parameters:** the first line shows the applied voltage in real time and the second line shows the generated internal reinforcement signal $\bar{r}$.

- **Output:** shows the trial number and the result condition (failure or success). The next two lines show the IAE and ITAE value criteria.

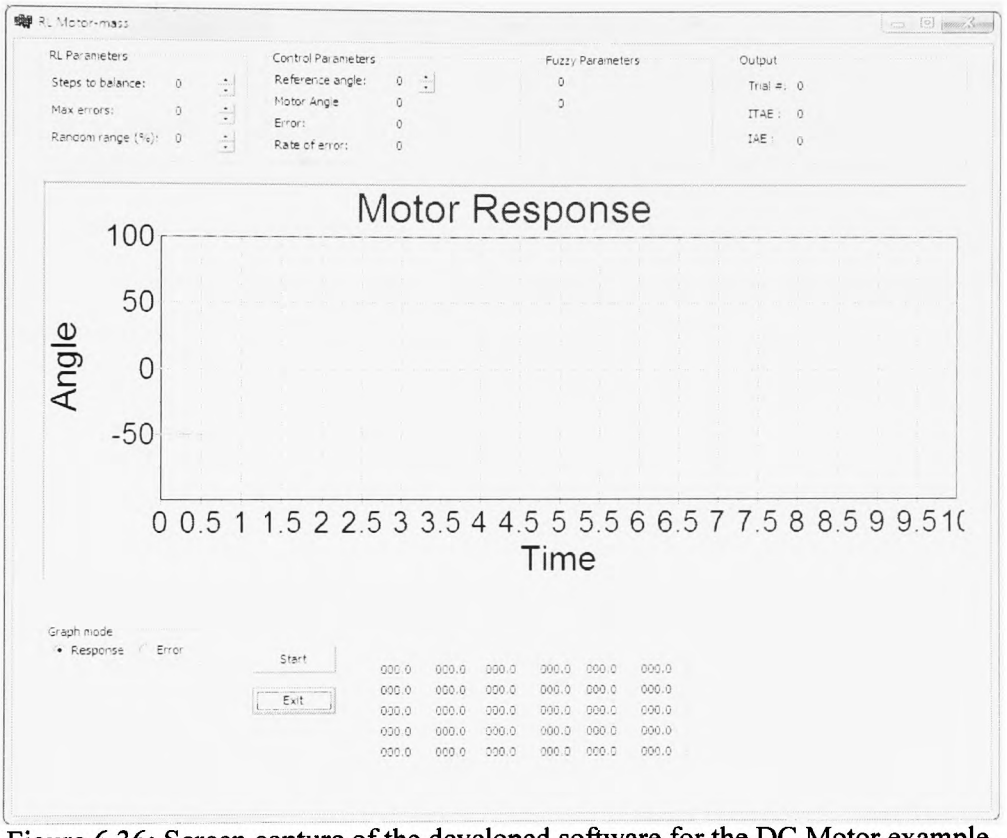

Figure 6.36: Screen capture of the developed software for the DC Motor example.

The corresponding relevant code is presented in Appendix C at the end of this dissertation. The complete code and a screen captured video are provided in the attached CD.

6.3.2.5 Results

The coded program search for possible probability values, and adjust the corresponding one, accordingly. As for all the experiments of this section, it is called "one test" to a set of trials performed by the agent until the probabilities of

success are learnt so that the system does not fail. A trial is defined as the period, in which a new policy is tested by the Actor, if a failure occurs, then the policy is updated and a new trial is started.

For this experiment two different tests were performed. The first test consisted on a step signal of 90deg as reference signal, after the learning is completed the signal is inverted (-90deg), so that the learning agent can explore different states, and then continues inverting every 1.5 seconds. This test can be run by executing the stand alone executable file RL_MOTOR_SS.exe found in X:\DC Motor Control Problem, where X is the CD drive letter. The results of this test can be seen in Figure 6.37 and Figure 6.38.

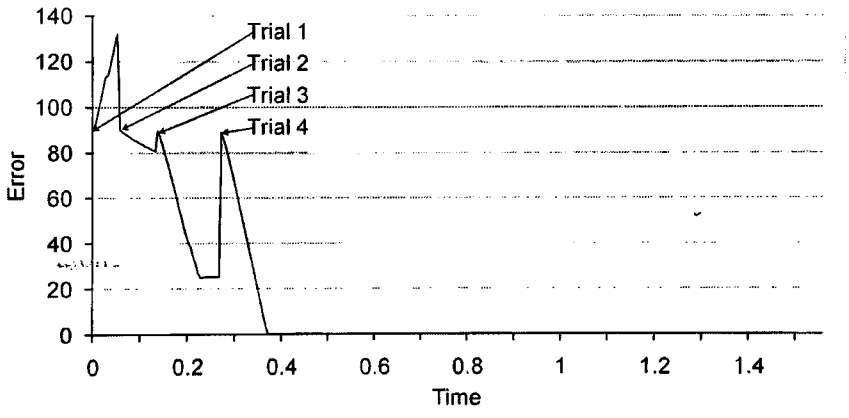

Figure 6.37: Trials for learning.

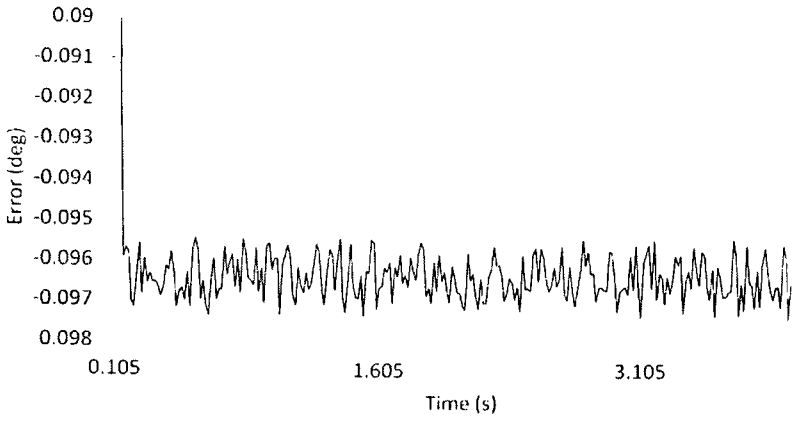

Figure 6.38: Motor error in steady state.

Figure 6.37 illustrates the learning process. The system starts 90° apart from the reference. After four trials, the agent configured the fuzzy logic rules so that the error rapidly converged to zero. Figure 6.38 shows the steady state error after a successful learning run. It can be observed the average error in steady state is around 0.0965deg or in percentage an error of 0.1% with no overshoot and a very fast rise time.

This test was performed twenty times; the agent was able to learn the probability values in four trials in every single test. The values for each probability after every test were observed to be, in every case, similar to the ones shown in Table 6.14.

TABLE 6.14 PROBABILITY MATRIX SHOWING THE PROBABILITY OF SUCCESS OF PERFORMING ACTION A1 FOR EVERY RULE.

$\dot{e}$ \ e	VLN	LN	SN	SP	LP	VLP
LN	0.49	0.49	0.49	0.49	0.50	0.50
SN	0.49	0.38	0.00	0.62	0.50	0.50
C	0.49	0.18	0.00	0.99	0.77	0.51
SP	0.47	0.07	0.07	0.93	0.92	0.53

LP	0.49	0.48	0.48	0.51	0.98	0.82

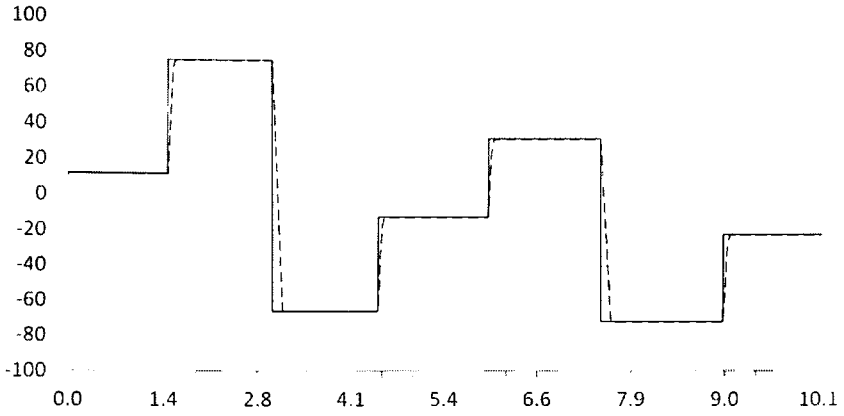

Figure 6.39: Motor response to a random step reference.

The second test consisted in presenting a square signal of random values. This test can be run by executing the stand alone executable file RL_MOTOR_RS.exe found in X:\DC Motor Control Problem, where X is the CD drive letter. Figure 6.39 shows the motor angle when a random step signal is used as reference.

In the second test, it was observed that the agent learns to control the motor in 10 trials for a positive signal and further 2 trials in average when the signal is inverted. This happens because the controller have already a good approximation of the control parameters around a state where the error is close to zero, so it requires less trials to complete the rule matrix of probabilities with the final values.

6.3.2.6 Conclusions

This section presented an experiment on a simulated DC motor for control using the proposed GPFRL algorithm. The key ingredient of this framework is a probabilistic model for the transition dynamics, as in the previous cases this experiment shown that the explicit incorporation of a probability term into the

decision making process provided a fast learning, flexibility and robustness in the configuration of a fuzzy inference system for optimal control. To our best knowledge, we can confirm an unprecedented speed of learning.

6.4 Summary

This chapter presented four experiments used to test different aspects of the proposed GPFRL method. The experiments performed were: random walk on a grid world, cart-pole balancing problem, DC motor control and mobile robot navigation. Results and conclusion have been detailed for each experiment.

The next chapter provides a final conclusion and observations and proposes some topics for future research.

Chapter 7

Concluding Discussion

7.1 Introduction

This work has explored the different advantages and limitations of three different popular paradigms, fuzzy logic systems, probabilistic theory and reinforcement learning separately. It was concluded that each of this paradigms could very well complement the drawbacks of the others and seamlessly work together in a cooperative rather than competitive way.

Fuzzy logic systems are good at generalizations, and due to its distinctive characteristics, it is able to handle non-statistical uncertainties, and fuzziness; however under certain conditions, the design and development of rather large or more complex systems can be too complicated for human operators.

Reinforcement learning methods have been an intense focus of research in the last decade. Research has proven that reinforcement learning can be successfully used in many different areas, such as decision making or control. A remarkable

characteristic is that RL methods do not require input-output pairs for training or previous knowledge of the environment model. RL only uses sparse signal information in order to reach to optimal conclusions. Therefore using RL for automatic tuning of fuzzy logic parameters have also being the focus of recent research.

Probabilistic theory is still one of the most effectives way (and most explored) to deal with uncertainties, especially stochastic uncertainty. The fusion of probabilistic theory with fuzzy logic controllers have shown to be a powerful tool for practical areas such as finance and weather forecasting. Both paradigms can work in collaboration, in order to complement each other. As a result, probabilistic fuzzy logic system can handle a very large range of uncertainties.

The present work combines these three paradigms into a novel method, able to learn optimal policies for control or decision making whilst being resistant to stochastic, non-stochastic, uncertainties, randomness and fuzziness. Four different experiments were carried on to ratify our claims; a classic cart-pole balancing problem, a controller for a DC motor, a random walk within a grid world and a windy grid world, and finally a real Khepera III mobile robot.

The rest of this chapter is organized as follows; Section two highlights the contributions of this dissertation. In section three some important observations are described. Section four proposes some future work and in section five the final conclusions are given.

7.2 Contributions of This Work

As a recall from the introduction, a list of the contributions of this work can be summarized in:

- This work presented a novel algorithm, able to find optimal policies under uncertainties in an automatic way.

- The algorithm described in this work proposed a modification to the actor critic reinforcement learning method where, the introduction of a probabilistic term in the learning provided a more effective way to deal with uncertainty while dramatically improving the performance of the system.

- Through several different experiments, the proposed method has demonstrated a high performance under situations of decision making and control.

- The observed results of this work showed that the GPFRL method can find its major application in the uncalibrated control of non-linear, multiple inputs, and multiple output systems, especially in situations with high uncertainty.

- The GPFRL algorithm was compared in a grid world, with the well-established RL methods, SARSA and Q-learning. In this experiment the GPFRL method showed not only a strong convergence but also a much faster one.

- GPFRL has learned to control the classically "hard" inverted pendulum system with a comparatively better performance than other methods in terms of learning speed and accuracy.

- The GPFRL proved to be a good method for optimizing control systems. This was tested in the DC motor example, where the proposed method was used to find the rule matrix of a fuzzy

logic controller, with the task of regulating the position of a motor shaft.

- The GPFRL method was implemented on a Khepera III robot with the task of avoiding obstacles. The results of this experiment were conclusive, a fast learning rate, and a statistical solution to the obstacle avoidance problem was found.

7.3 Observations

This section presents some observations gathered during the development of this work.

- Through the process of design and evaluation of the presented method it was observed that the selection of a n appropriate "reinforce" signal becomes crucial for the success of the learning process. In reinforcement learning, the agent is not aware of the system it is working on, nor the objective it have, it only works on the basis of a "success" or "failure" signal. Therefore, the appropriate selection of a reinforce signal becomes, in other words, what is it to be considered success or failure. This question might become obvious on simple linear systems, but with more complex systems and tasks, the definition or generation of this signal can be of high complexity.

- Some interesting results were observed along the testing of more complex systems, like the Khepera III mobile robot. Having in mind that the purpose of the reinforcement learning algorithm was to learn how to avoid obstacles, some interesting effects were observed:

- The learning agent "concluded" that if the robot does not move, it will not crash, therefore after crashing for the first time in one experiment the robot stop moving.

- The learning agent "concluded" that if the robot spins over its own axis, it will not crash, therefore after crashing for the first time in another experiment the robot started spinning.

- After including an algorithm that promotes forward motion of the robot, the learning agent "concluded" that if the robot moves forward and backwards only a small distance, it will not crash, therefore after crashing for the first time in one experiment the robot presented the described behaviour.

- The above observations presented a real challenge at the time of designing an experiment. All of the above cases can be considered a successful learning from the RL point of view, although the final behaviours were completely unpredictable and diverse. This suggests that care must be taken when designing systems with learning agents, where the selection of actions can lead to undesired behaviours despite of a successful learning.

7.4 Future Work

Nothing is perfect and no work can ever be complete, there will always be tradeoffs, and this work is no exception. Therefore it is suggested some future

improvements based on the shortcomings of this proposed algorithm. These suggestions are divided in two groups, theoretical and practical suggestions.

7.4.1 Theoretical Suggestions

- Learning still is a black box in most of the cases. The presence of uncertainty and the concept of knowledge itself make it intrinsically difficult if not impossible to model and analyze the final behaviour of a learning system; therefore no system with unsupervised learning should be unsupervised. It is suggested the research of methods combining the probability estimation of reinforcement learning, as the one proposed in this work, with other methods that can incorporate the knowledge of human operators; especially for situation where the result of selecting actions or beha viours with a relativ ely "high" pr obability of success can have considerable consequences. As an example, it can be suggested the use of prospect theory, which is a method of calculating decisions not only based on probabilities of success but also based on a risk evaluation.

- For cases were the training is episodic, the study of a way of storing the rewards obtained during an episode so that they can be backward updated could allow a better and more efficient updating, of course, at the expense of requiring more memory.

- In this work, the fuzzy inference systems have been manually configured with good results. Nevertheless, the implementation of structure/parameter learning can further improve the

performance of the controller and reduce the tuning phase time considerably.

7.4.2 Practical Suggestions

- Although mathematical models of dynamical systems can be accurate to a high degree, it doesn't come close to real systems in terms of uncertainty. Therefore, it is highly suggested a more rigorous testing of the proposed GPFRL method on systems interacting with real environments, such as with mobile robots navigating on uncharted environments, highly nonlinear manipulators, etc.

- Moreover, it is suggested the exploration on how this proposed algorithm performs on more challenging tasks, especially in systems with higher dimensional states.

7.5 Final Conclusions

It is my belief after observing the results of the proposed algorithm, that the objectives of this dissertation have been successfully met. I can also anticipate the effective solution to more complex problems using this algorithm. As any approach, there are limitations and shortcomings in the proposed algorithm, which should be improved further.

REFERENCES

AKERKAR, R. 2005. *Introduction to artificial intelligence*, PHI Learning Pvt. Ltd.

ALMEIDA, R. J. & KAYMAK, U. 2009. Probabilistic fuzzy systems in value-at-risk estimation. *International Journal of Intelligent Systems in Accounting, Finance and Management,* 16, 49-70.

AN, C. H., ATKESON, C. G. & HOLLERBACH, J. 1988. *Model-based control of a robot manipulator,* Boston, USA, MIT Press.

ANDERSON, C. W. 1986. Learning and problem-solving with multilayer connectionist systems. University of Massachusetts.

BACON, A. 2009. *Vagueness and uncertainty.* BPhil, University of Oxford.

BAIRD, L. C. 1999. *Reinforcement learning through gradient descent.* Doctor of Philosophy, Carnegie Mellon University.

BAIRD, L. C. & MOORE, A. 1999. Gradient descent for general reinforcement learning. *Advances in Neural Information Processing Systems 2.* MIT Press.

BARRICELLI, N. 1954. Esempi numerici di processi di evoluzione. *Methodos,* 45-68.

BARTO, A. G. & ANANDAN, P. 1985. Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man and Cybernetics,* SMC-15, 360-374.

BARTO, A. G. & JORDAN, M. I. 1987. Gradient following without backpropagation in layered networks. *IEEE First Annual International Conference on Neural Networks,* 2, 629-636.

BARTO, A. G., SUTTON, R. S. & A NDERSON, C. W. 1983. Neuron-like adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics,* 13, 835-846.

BELLMAN, R. E. 1957. *Dynamic programming,* Princeton, New Jersey, Princeton University Press.

BERENJI, H. R. 1996. Fuzzy Q-learning for generalization of reinforcement learning. *5th IEEE International Conference on Fuzzy Systems.* New Orleans, Louisiana, USA.

BERENJI, H. R. & KHEDKAR, P. 1992. Learning and tuning fuzzy logic controllers through reinforcements. *IEEE Transactions on Neural Networks,* 3, 724-740.

BERENJI, H. R. & KHEDKAR, P. S. 1998. Using fuzzy logic for performance evaluation in reinforcement learning. 18, 131-144.

BERG, J. V. D., KAYMAK, U. & BERG, W.-M. V. D. 2004. Financial markets analysis by using a probabilistic fuzzy modelling approach. *International Journal of Approximate Reasoning,* 35, 291-305.

BERG, J. V. D., KAYMAK, U. & BERGH, W.-M. V. D. 2002. Fuzzy classification using probability-based rule weighting. *IEEE International Conference on Fuzzy Systems.* Honolulu, HI, USA.

BLACKMORE, L. 2006. A probabilistic particle control approach to optimal, robust predictive control. *AIAA Guidance, Navigation and Control Conference.*

CHEESEMAN, P. 1985. In defense of probability. *9th International Joint Conference on Artificial Intelligence.* Los Angeles, California: Morgan Kaufmann Publishers Inc.

CLEVELAND, W. S. 1979. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association,* 74, 829-836.

COMON, P. 1994. Independent component analysis: a new concept? *Signal Processing,* 36, 287-314.

DAW, N. D. 2003. Reinforcement learning models of the dopamine system and their behavioral implications. PhD, Carnegie Mellon University.

DAYAN, P. & DAW, N. D. 2008. Decision theory, reinforcement learning, and the brain. *Cognitive, Affective, & Behavioral Neuroscience,* 8, 429-453.

DEARDEN, R., FRIEDMAN, N. & RUSSELL, S. 1998. Bayesian Q-learning. *Proceedings of the 15th national conference on Artificial intelligence.* CA, USA: AAAI Press.

DEISENROTH, M. P. & RASMUSSEN, C. E. 2010. Reducing model bias in reinforcement learning. *Workshop on Learning and Planning from Batch Time Series Data.* Whistler, BC, Canada.

DEMSPSTER, A. P. 1969. Upper and lower probability inferences for families of hypotheses with monotone density rations. *Annals of Mathematical Statistics,* 40, 953-969.

DORIGO, M. & COLOMBETTI, M. 1998. *Robot shaping: an experiment in behavior engineering,* Cambridge, Massachusetts, The MIT Press.

DOYA, K. 2002. Metalearning and neuromodulation. *Neural Networks,* 15, 495-506.

DRIANKOV, D., HELLENDOORN, H. & REINFRANK, M. 1996. *An introduction to fuzzy control,* New York, Springer-Verlag.

DUBOIS, D. 2006. Possibility theory and statistical reasoning. *Computational Statistics & Data Analysis,* 51, 47-69.

DWIVEDI, A., MISHRA, D. & KALRA, P. K. 2006. Handling uncertainties using probability theory to possibility theory. *Directions Magazine.* Kanpur, India: India Institute of Technology.

FIX, E. & HODGES, J. L. 1951. Discriminatory analysis, nonparametric discrimination: consistency properties. California, USA: California University at Berkley.

FOGEL, L. J., OWENS, A. J. & WALSH, M. J. 1966. *Artificial intelligence through simulated evolution,* John Wiley & Sons.

FRANK, M. J. & CLAUS, E. D. 2006. Anatomy of a decision: striato-orbitofrontal interactions in reinforcement learning, decision making, and reversal. *Psychological Review,* 113, 300–326.

FULLER, R. 2000. *Introduction to neuro-fuzzy systems,* Berlin, Physica-Verlag HD.

GARDIOL, N. H. & KAELBLING, L. P. 2006. Computing action equivalences for planning under time-constraints. *Learning and Intelligent Systems.* Cambridge, Massachusetts, USA: Massachusetts Institute of Technology.

GASKETT, C. 2003. Reinforcement learning under circumstances beyond its control. *International Conference on Computational Intelligence for Modelling Control and Automation* Vienna, Austria.

GASKETT, C., FLETCHER, L. & ZELINSKY, A. 2000. Reinforcement learning for a vision based mobile robot. *International Conference on Intelligent Robots and Systems.*

GLORENNEC, P. Y. 2000. Reinforcement learning: an overview. *European Symposium on Intelligent Techniques.* Aachen, Germany.

GLORENNEC, P. Y. & JOUFFE, L. 1997. Fuzzy Q-Learning. *Procedings of The IEEE International Conference on Fuzzy Systems.* Barcelona, Spain.

GOLDBERG, D. E. 1994. Genetic and evolutionary algorithms come of age. *Communications of the ACM,* 37, 113-120.

HEBB, D. O. 1949. The Organization of Behavior: A Neuropsychological Theory, New York, USA, Wiley.

HINOJOSA, W., NEFTI, S., GRAY, J. & KAYMAK, U. 2008. Reinforcement learning for probabilistic fuzzy controllers. *International Conference on Control.* Manchester, UK.

HINTON, G.E. 1989. Connectionist learning procedures. *Artificial Intelligence,* 40, 185-234.

HOLLAND, J. H. 1992. Genetic algorithms. *Scientific American Magazine,* 267, 66-73.

JANG, J.-S. R. 1993. ANFIS: Adaptive-Network-based Fuzzy Inference Systems. *IEEE Transactions on Systems, Man and Cybernetics,* 23, 665-685.

JANG, J.-S. R., SUN, C.-T. & MIZUTANI, E. 1997. *Neuro-fuzzy and soft computing*, Prentice Hall.

JAULMES, R., PINEAU, J. & PRECUP, D. 2005. Probabilistic robot planning under model uncertainty: an active learning approach. *In:* GRUDIC, G. & MULLIGAN, J. (eds.) *NIPS Workshop on Machine Learning Based Robotics in Unstructured Environments.* Whistler, Canada.

JOUFFE, L. 1998. Fuzzy inference system learning by reinforcement methods. *IEEE Transactions on Systems, Man, and Cybernetics,* 28, 338-355.

KALYANAKRISHNAN, S. & STONE, P. 2009. An empirical analysis of value function-based and policy search reinforcement learning. *8th International Conference on Autonomous Agents and Multiagent Systems.* Budapest, Hungary: International Foundation for Autonomous Agents and Multiagent Systems.

KANTARDZIE, M. 2002. Data mining: concepts, models, methods, and algorithms, WileyBlackwell.

KOHONEN, T. 1988. Self-organization and associative memory, Berlin, Springer.

KOSKO, B. 1991. Neural networks and fuzzy systems: a dynamical systems approach to machine intelligence, Prentice Hall.

LA, B. M. 2003. Mobile robot navigation using fuzzy reinforcement learning. *Proceedings of The 9th National UROP Congress.*

LANGLEY, P. 1995. *Elements of machine learning,* San Francisco, CA, Morgan Kaufmann.

LAUD, A. & DEJONG, G. 2003. The influence of reward on the speed of reinforcement learning: an analysis of shaping. *Proceedings of The 20th International Conference on Machine Learning (ICML)*, 440–447.

LAURITZEN, S. L. & SPIEGELHALTER, D. J. 1988. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society B,* 50, 157-224.

LAVIOLETTE, M. & SEAMAN, J. W. 1994. Unity and diversity of fuzziness-from a probability viewpoint. *IEEE Transactions on Fuzzy Systems,* 2, 38-42.

LEE, C.-C. 1991. A self-learning rule-based controller employing approximate reasoning and neural net concept. *International Journal of Intelligent Systems,* 6, 71-93.

LIANG, P. & SONG, F. 1996. What does a probabilistic interpretation of fuzzy sets mean? *IEEE Transactions on Fuzzy Systems,* 4, 200-205.

LIN, C.-J. & LIN, C.-T. 1996. Reinforcement learning for an ART-based fuzzy adaptive learning control network. *IEEE Transactions on Neural Networks,* 7, 709-731.

LIN, C.-J. & XU, Y.-J. 2006. A novel genetic reinforcement learning for nonlinear fuzzy control problems. *Neurocomputing,* 69, 2078-2089

LIN, C.-K. 2003. A reinforcement learning adaptive fuzzy controller for robots. *Fuzzy Sets and Systems,* 137, 339-352.

LIN, C.-T. 1995. A neural fuzzy control system with structure and parameter learning *Fuzzy Sets and Systems,* 70, 183-212.

LIN, C.-T. & LEE, C. S. G. 1993. Reinforcement structure/parameter learning for neural-network-based fuzzy logic control systems. *IEEE International Conference on Fuzzy Systems.* San Francisco, CA, USA.

LIN, C.-T. & LEE, C. S. G. 1994. Reinforcement structure/parameter learning for neural-network-based fuzzy logic control systems. *IEEE Transactions on Fuzzy Systems,* 2, 46-63.

LITTMAN, M. L. 2001. Value-function reinforcement learning in Markov games. *Journal of Cognitive Systems Research 2,* 2, 55-66.

LIU, Z. & LI, H.-X. 2005. A probabilistic fuzzy logic system for modelling and control. *IEEE Transactions on Fuzzy Systems,* 13, 848-859.

MAMDANI, E. & ASSILIAN, S. 1975. An experiment in linguistic synthesis with a fuzzy logic controller. *International Journal of Man-Machine Studies,* 7, 1-13.

MARTHI, B. 2007. Automatic shaping and decomposition of reward functions. *In:* GHAHRAMANI, Z. (ed.) *Proceedings of The 24th International Conference on Machine Learning* Corvallis, Oregon, USA: ACM.

MATARIC, M. J. 1994. Reward functions for accelerated learning. Proceedings of The 11th International Conference on Machine Learning (ICML), 181-189.

MCNEILL, F. M. & THRO, E. 1994. *Fuzzy logic: a practical approach,* Morgan Kaufmann Pub.

MEGHDADI, A. H. & AKBARZADEH-T., M.-R. 2001. Probabilistic fuzzy logic and probabilistic fuzzy systems. *10th IEEE International Conference on Fuzzy Systems.* Melbourne, Australia.

MEGHDADI, A. H. & AKBARZADEH-T., M.-R. 2003. Uncertainty modeling through probabilistic fuzzy systems. *4th International Symposium on Uncertainty Modelling and Analysis.* College Park, Maryland, USA.

MICHALEWICZ, Z. 1996. Genetic algorithms + data structures = evolution programs, London, UK, Springer-Verlag.

MITCHELL, T. 1997. *Machine learning,* McGraw-Hill.

MITCHELL, T. M., KELLER, R. M. & KEDAR-CABELLI, S. T. 1986. Explanation-based generalization: a unifying view. *Machine Learning,* 1, 47-80.

MOORE, A. W. & ATKESON, C. G. 1993. Prioritized sweeping: reinforcement learning with less data and less real time. *Machine Learning,* 13, 103-130.

MORIARTY, D. E., SCHULTZ, A. C. & GREFENSTETTE, J. J. 1999. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research,* 11, 241-276.

MUSTAPHA, S. M. & LACHIVER, G. 2000. A modified actor-critic reinforcement learning algorithm. *Canadian Conference on Electrical and Computer Engineering.* Halifax Nova Scotia Canada.

NAUCK, D., KLAWONN, F. & KRUSE, R. 1997. *Foundations of neuro-fuzzy systems,* New York, John Wiley & Sons.

NG, K. C. & LI, Y. 1994. Design of sophisticated fuzzy logic controllers using genetic algorithms. *3rd IEEE International Conference on Fuzzy Systems.* Orlando, FL , USA.

NIV, Y. 2009. Reinforcement learning in the brain. *Journal of Mathematical Psychology,* 53, 139-154.

PAN, Y., KLIR, G. J. & YUAN, B. 1996. Bayesian inference based on fuzzy probabilities.

PARK, J.-J., KIM, J.-H. & SONG, J.-B. 2007. Path planning for a robot manipulator based on probabilistic roadmap and reinforcement learning. *International Journal of Control, Automation, and Systems,* 5, 674-680.

PEARL, J. 1982. Reverend Bayes on inference engines: a distributed hierarchical approach. *American Association of Artificial Intelligence National Conference on AI.* Pittsburgh, Pennsylvania, USA.

PEARL, J. 1988. Probabilistic reasoning in intelligent systems: networks of plausible inference, San Mateo, California, USA, Morgan Kaufmann Publishers, Inc.

PEDNAULT, E., ABE, N. & ZADROZNY, B. 2002. Sequential cost-sensitive decision making with reinforcement learning. *8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 259-268.

PINEAU, J., GORDON, G. & THRUN, S. 2003. Point-based value iteration: an any time algorithm for POMDPs. *International Joint Conference on Artificial Intelligence (IJCAI).* Acapulco, Mexico.

POUPART, P. & BOUTILIER, C. 2004. VDCBPI: an approximate scalable algorithm for large POMDPs. *Advances in Neural Information Processing Systems.* Vancouver, British Columbia, Canada.

POUPART, P., VLASSIS, N., HOEY, J. & REGAN, K. 2006. An analytic solution to discrete Bayesian reinforcement learning. *Proceedings of the 23rd International Conference on Machine Learning.* New York, NY, USA: ACM.

PROROK, A., ARFIRE, A., BAHR, A., FARSEROTU, J. R. & MARTINOLI, A. 2010. Indoor Navigation Research with the Khepera III Mobile Robot: An Experimental Baseline with a Case-Study on Ultra-Wideband Positioning. *International Conference on Indoor Positioning and Indoor Navigation.* ETH Zurich, Switzerland.

QUINLAN, J. R. 1986. Induction of decision trees. *Machine Learning,* 1, 81-106.

REGAN, H. M., COLYVAN, M. & BURGMAN, M. A. 2000. A proposal for fuzzy International Union for the Conservation of Nature (IUCN) categories and criteria. *Biological Conservation,* 92, 101-108.

ROGOVA, G., SCOTT, P. & LOLETT, C. 2002. Distributed reinforcement learning for sequential decision making. *5th International Conference on Information Fusion.*

ROSS, T. 2004. Fuzzy logic with engineering applications, John Wiley & Sons Ltd.

ROY, N., GORDON, G. & THRUN, S. 2005. Finding approximate POMDP solutions through belief compression. *Journal of Artificial Intelligence Research,* 23, 1-40.

RUMELHART, D. E., HINTON, G. E. & WILLIAMS, R. J. 1986. Learning internal representations by error propagation. *Parallel distributed processing: explorations in the microstructure of cognition.* MIT Press.

RUMMERY, G. A. & NIRANJAN, M. 1994. On-line q-learning using connectionist systems. Cambridge: Cambridge University.

RUSSELL, S. & NORVIG, P. 1994. *Artificial intelligence: a modern approach*, Prentice Hall.

SAADE, J. J. 2011. Fuzzy sets and inference as an effective methodology in the construction of intelligent controllers *Transactions on Internet Research* 7, 3-17.

SAINSBURY, R. M. 1995. *Paradoxes*, Cambridge University Press.

SCHAPIRE, R. E. 1990. The Strength of Weak Learnability. *Machine Learning,* 5, 197-227.

SCHULTZ, W., DAYAN, P. & MONTAGUE, P. R. 1997. A neural substrate of prediction and reward. *Science Magazine,* 275, 1593 - 1599.

SHAFER, G. 1976. *A mathematical theory of evidence,* Princeton, New Jersey, Princeton University Press.

SILER, W. & BUCKLEY, J. J. 2004. *Fuzzy expert systems and fuzzy reasoning,* New Jersey, Wiley-Interscience.

SINGH, S., JAAKKOLA, T., LITTMAN, M. L. & ARI, C. S. 2000. Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. *Machine Learning,* 38.

STRENS, M. 2000. A Bayesian framework for reinforcement learning. *Proceeedings of the 17th International Conference on Machine Learning.* Stanford University, California: ICML.

SUTTON, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine Learning,* 3, 9-44.

SUTTON, R. S. & BARTO, A. G. 1981. Toward a modern theory of adaptive networks: expectation and prediction. *Psychological Review,* 88, 135-170.

SUTTON, R. S. & BARTO, A. G. 1987. A temporal-difference model of classical conditioning. *9th Annual Conference of the Cognitive Science Society.* Seattle,WA.

SUTTON, R. S. & BARTO, A. G. 1998. *Reinforcement learning: an introduction,* Cambridge, MA, MIT Press.

TANAKA, K. & WANG, H. O. 2001. Fuzzy control systems design and analysis: a linear matrix inequality approach, New York, John Wiley & Sons, Inc.

TANNERT, C., ELVERS, H.-D. & JANDRIG, B. 2007. The ethics of uncertainty. In the light of possible dangers, research becomes a moral duty. *EMBO Reports,* 8, 892-896.

THRUN, S. 2000. Probabilistic algorithms in robotics. *AI Magazine,* 21, 93-109.

THRUN, S., BEETZ, M., BENNEWITZ, M., BURGARD, W., CREMERS, A. B., DELLAERT, F., FOX, D., HÄHNEL, D., ROSENBERG, C., ROY, N., SCHULTE, J. & SCHULZ, D. 2000. Probabilistic algorithms and the interactive museum tour-guide robot Minerva. *International Journal of Robotics Research,* 19, 972-998.

VALAVANIS, K. P. & SARIDIS, G. N. 1991. Probabilistic modelling of intelligent robotic systems. *IEEE Transactions on Robotics and Automation,* 7, 164-171.

VALIANT, L. G. 1984. A theory of the learnable. *Communications of the ACM Magazine,* 27, 1134-1142.

WANG, X.-S., CHENG, Y.-H. & YI, J.-Q. 2007. A fuzzy actor-critic reinforcement learning network. *Information Sciences,* 177, 3764-3781.

WATKINS, C. J. C. H. 1989. *Learning from delayed rewards.* Doctor of Philosophy, King's College.

WATKINS, C. J. C. H. & DAYAN, P. 1992. Q-learning. *Machine Learning,* 8, 279-292.

WILLGOSS, R. A. & IQBAL, J. 1999. Reinforcement learning of behaviours in mobile robots using noisy infrared sensing. *Australian Conference on Robotics and Automation.* Brisbane, Australia.

WILLIAMS, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning,* 8, 229-256.

ZADEH, L. A. 1965. Fuzzy sets. *Information and Control,* 8, 338-353.

ZADEH, L. A. 1973. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Transactions on Systems, Man, and Cybernetics,* 3, 28-44.

ZADEH, L. A. 1978. Fuzzy sets as a basis for theory of possibility. *Fuzzy Sets and Systems,* 1, 3-28.

ZADEH, L. A. 1984. Fuzzy probabilities. *Information Processing and Management,* 20, 363-372.

ZADEH, L. A. 1995. Probability theory and fuzzy logic are complementary rather than competitive. *Technometrics,* 37, 271-276.

ZARANDI, M. H. F., JOUZDANI, J. & ZARANDI, M. F. 2008. Reinforcement learning for fuzzy control with linguistic states. *Journal of Uncertain Systems,* 2, 54-66.

Appendix

Appendix A

Random Walk Problem

SARSA

```
%-------------------------------------
% Constants
%-------------------------------------

% Size of the Grid
gridSize = 10;

% Location of reward
rewardM = ceil(gridSize/2);
rewardN = rewardM+1;

% Location to start from
startM = 1;
startN = 1;

% Parameters
alpha    = 0.4; % Step-size
epsilon   = 0.01; % e-Greedy behaviour for maximum stability
gamma    = 0.95; % Discount
totalPlays   100; % Number of plays to average over
totalTrials = 40;  % Trials per play

% Total # steps for each trial over all plays
totalNumSteps = zeros(totalTrials, 1);

% Total reward values over all plays
totalReward = zeros(totalTrials, 1);

%-------------------------------------
% Start walking   SARSA
%-------------------------------------
for j = 1 : totalPlays
  % Store # steps for each trial
  numSteps = zeros(totalTrials, 1);

  % Initialize V(s)
  grid = zeros(gridSize, gridSize);
  grid(rewardM, rewardN) = 1;

  for i = 1   totalTrials
    % Current position on grid is represented by m,n
    m = startM;
    n = startN;
    prevM = startM;
    prevN = startN;

    % Begin stepping through a path
    found = 0;
    while(found == 0)
      % increment # steps taken this trial
      numSteps(i) = numSteps(i) + 1;

      % pick an action and avoid going over grid borders
      N = m-1;
      if(N < 1)
        N = 1;
```

```
      end
      E   n+1;
      if(E > gridSize)
        E = gridSize;
      end
      S = m+1;
      if(S > gridSize)
        S = gridSize;
      end
      W = n-1;
      if(W < 1)
        W = 1;
      end
      choices   [ grid(N,n), grid(m,E), grid(S,n), grid(m,W) ];

      % choose the path with the maximum value, or explore
      indices = find(choices == max(choices));
      maximum = max(choices);
      if((length(indices) == 1) && (rand(1)   epsilon >= 0))
        index = indices(1);
      else
        index = ceil(length(choices) * rand(1)); % randomly select direction
      end

      %Take action
      if(index == 1)   % grid(m-1,n)
        m = N;
      elseif(index == 2) % grid(m,n+1)
        n = E;
      elseif(index == 3) % grid(m+1,n)
        m = S;
      elseif(index == 4) % grid(m,n-1)
        n = W;
      else
        disp('error');
      end

      % stop after this step if the reward was found
      if(m == rewardM && n == rewardN)
        found = 1;
      end

      %Expected discounted reward
      expDiscReward = gamma*grid(m,n);

      % calculate prediction error
      predErr = expDiscReward   grid(prevM,prevN);

      % V(s) = V(s) + alpha[r' + V(s')   V(s)]
      grid(prevM,prevN) = grid(prevM,prevN) + alpha*predErr;

      % update values for next step, and directional choices
      prevM = m;
      prevN = n;
    end
  end

  % Update storage matrices for across all plays
  totalNumSteps = totalNumSteps + numSteps;
end

%-------------------------------------
% Outputs
%-------------------------------------
% Average number of steps across all plays
plot(totalNumSteps / totalPlays, 'color', 'k', 'LineStyle', '-');
```

Q-LEARNING

```
%-------------------------------------
% Constants
%-------------------------------------

% Size of the Grid
gridSize = 10;

% Location of reward
rewardM = ceil(gridSize/2);
rewardN = rewardM+1;

% Location to start from
startM = 1;
startN = 1;

% Parameters
alpha      0.4; % Step-size
epsilon    = 0.01; % e-Greedy behaviour for maximum stability
gamma      0.95; % Discount
totalPlays = 100; % Number of plays to average over
totalTrials = 40;  % Trials per play

% Total # steps for each trial over all plays
totalNumSteps = zeros(totalTrials, 1);

% Total reward values over all plays
totalReward = zeros(totalTrials, 1);
averageReward = zeros(totalTrials, 1);

%-------------------------------------
% Start walking   Q-Learning
%-------------------------------------
for j = 1 : totalPlays
  % Store # steps for each trial
  numSteps = zeros(totalTrials, 1);

  % Initialize V(s)
  grid = zeros(gridSize, gridSize);
  grid(rewardM, rewardN)   1;

  for i = 1   totalTrials
    % Current position on grid is represented by m,
    m   startM;
    
    startN;
    prevM = startM;
    prevN = startN;

    % Begin stepping through a path
    found = 0;
    while(found == 0)
      % increment # steps taken this trial
      numSteps(i) = numSteps(i) + 1;

      % pick an action and avoid going over grid borders
      N = m-1;
      if(N < 1)
```

```
        N = 1,
      end
      E = n+1;
      if(E > gridSize)
        E = gridSize;
      end
      S = m+1;
      if(S > gridSize)
        S = gridSize;
      end
      W = n-1;
      if(W < 1)
        W   1;
      end
      choices    [ grid(N,n), grid(m,E), grid(S,n), grid(m,W) ];

      % choose the path with the maximum value, or explore
      indices    find(choices == max(choices));
      maximum    max(choices);
      if((length(indices) == 1) && (rand(1)    epsilon >= 0))
        index = indices(1);
      else
        index = ceil(length(choices) * rand(1)); % randomly select direction
      end

      %Take action
      if(index == 1)    % grid(m-1,n)
        m = N;
      elseif(index == 2) % grid(m,n+1)
        n = E;
      elseif(index == 3) % grid(m+1,n)
        m = S;
      elseif(index == 4) % grid(m,n-1)
        n = W;
      else
        disp('error');
      end

      % stop after this step if the reward was found
      if(m == rewardM && n == rewardN)
        found = 1,
      end

      %Expected discounted reward
      expDiscReward = gamma*maximum;

      % calculate prediction error
      predErr = expDiscReward    grid(prevM,prevN);

      % V(s)    V(s) + alpha[r' + V(s')    V(s)]
      grid(prevM,prevN) = grid(prevM,prevN) + alpha*predErr;

      % update values for next step, and directional choices
      prevM    m;
      prevN = n;
    end
  end

  % Update storage matrices for across all plays
  totalNumSteps = totalNumSteps + numSteps;
end

%-------------------------------------
% Outputs
%-------------------------------------
% Average number of steps across all plays
plot(totalNumSteps / totalPlays, 'color', 'k', 'LineStyle', '-');
```

GPFRL

```
%-------------------------------------
% Constants
%-------------------------------------

% Parameters
Alpha          = 0.003;              % Actor learning rate
beta           = 0.005;              % Critic learning rate
epsilon        = 0.01;               % Exploration/explotation control parameter
gamma          = 0.95;               % Discount factor
totalActions   = 4;                  % Number of possible actions
totalAgents    = totalActions;       % Total number of learning agents

% Simulation control
gridSize    = 10;                    % Size of the Grid
totalStates = gridSize*gridSize;     % Total number of system states
rewardM     = ceil(gridSize/2);      % Location of reward in M axe
rewardN     = rewardM + 1;           % Location of reward in N axe
startM      = 1;                     % Location to start from in M axe
startN      = 1;                     % Location to start from in N axe
totalPlays  = 100;                   % Number of plays to average over
totalTrials = 40;                    % Trials per play

% Variables
totalNumSteps  = zeros(totalTrials, 1);% Total # steps for each trial over all plays
totalReward    = zeros(totalTrials, totalAgents);

%-------------------------------------
% Start walking - GPFRL
%-------------------------------------
for j = 1 : totalPlays
  % Store # steps for each trial
  numSteps = zeros(totalTrials, 1);

  % Initializing actor and critic weights
  v = zeros(totalStates, totalAgents);
  w = zeros(totalStates, totalAgents);

  for i = 1 : totalTrials
    disp(j+(i/100))
    % Current position on grid is represented by m,n
    m = startM;
    n = startN;
    prevM = startM;
    prevN = startN;

    % Clean some variables
    p  = zeros(totalAgents,1);
    oldp   p;

    % Begin stepping through a path
    found = 0;
    while(found == 0)
      % Increment # steps taken this trial
      numSteps(i) = numSteps(i) + 1;

      % Find probabilities from actor weights
```

```
sumWeights = zeros(totalAgents,1);
ro   zeros(totalStates, totalAgents);
for agent = 1 : totalAgents
  for state = 1 . totalStates
    ro(state, agent)  = logsig(w(state, agent));
    sumWeights(agent)    sumWeights(agent) + ro(state, agent);
  end
end

% Perform an observation (using boxes system)
stidx = ((m   1) * gridSize) + n;
X     zeros(totalStates, 1);
X(stidx) = 1;

% Finding final probability
P = zeros(totalAgents, 1);
for agent = 1   totalAgents
  P(agent) = ro(stidx, agent);
end

% Creating action set avoiding grid borders
N = m   1;
if(N < 1)
  N = 1;
end
E = n + 1;
if(E > gridSize)
  E = gridSize;
end
S   m + 1;
if(S > gridSize)
  S = gridSize;
end
W   n   1;
if(W < 1)
  W = 1;
end

% Initializing values
Nu  = zeros(totalAgents, 1);
stdev = zeros(totalAgents, 1);

% Stochastic random number generator
for agent   1   totalAgents
  stdev(agent) = 2*logsig(p(agent))-1;
  Nu(agent) = stdev(agent)*randn;
  if Nu(agent) >= 1
    Nu(agent) = 1;
  elseif Nu(agent) <= -1
    Nu(agent) = -1;
  end
end

% Finding choices (Probability + explotation/exploration)
choices = zeros(totalAgents, 1);
for agent=1:totalAgents
  choices(agent) = P(agent)+Nu(agent);
end

% Choose the path with the maximum probability, or explore
indices = find(choices == max(choices));
maximum = max(choices);
if((length(indices) == 1) && (maximum >= epsilon))
  index = indices(1);
else
  index = ceil(length(choices) * rand), % randomly select direction
end

%Take action
```

```
      r=zeros(totalAgents,1);
      if(index == 1)    % grid(m-1,n)
        m = N;
        r(index) = sign(m-rewardM);
      elseif(index == 2) % grid(m,n+1)
        n   E;
        r(index) = sign(rewardN-n);
      elseif(index == 3) % grid(m+1,n)
        m = S;
        r(index) = sign(rewardM-m);
      elseif(index == 4) % grid(m,n-1)
        n = W;
        r(index) = sign(n-rewardN);
      else
        disp('error random number too big');
      end

      % Stop after this step if the reward was found
      if(m == rewardM && n == rewardN)
        found  = 1;
      end

      % Saving prediction information
      oldp = p;
      p  = zeros(totalAgents,1);

      % Computing the prediction of eventual reinforcement
      for agent = 1:totalAgents
        p(agent) = v(stidx,agent);
      end

      % Computing the prediction error using temporal differences method
      rbar=zeros(totalAgents,1);
      for agent = 1:totalAgents
        predErr = gamma*p(agent)   oldp(agent);
        rbar(agent)   r(agent) + predErr;
      end

      % Learning thev value functions
      for agent = 1 : totalAgents
        for state = 1 : totalStates
w( state, agent ) = w( state, agent ) + alpha * rbar( agent ) * X( state ) * ( 1 /ro (
state, agent ) ) * exp( -w( state, agent ) ) * sumWeights( agent );
          v(state,agent)=v(state,agent)-beta*gamma*rbar(agent)*X(state);
        end
      end

      % update values for next step, and directional choices
      prevM   m;
      prevN   n;
    end
  end

  % Update storage matrices for across all plays
  totalNumSteps = totalNumSteps + numSteps;
end

%-------------------------------------
% Outputs
%-------------------------------------
% Average number of steps across all plays
plot(totalNumSteps / totalPlays, 'color', 'k', 'LineStyle', '-');
```

Appendix B

Cart-Pole Problem

MAIN PROGRAM

```
// RL_Prob_Cart_Pole.cpp   Defines the entry point for the console application
#include "stdafx.h"
#include "Mathrl.h"
#include "Cartpole.h"
#include "RLearning.h"
#include "Fuzzy.h"
#include "FilePrint.h"
#include <math.h>
#include "Rgenerator.h"
//--------------------------------------------------------------------------

#define MAX_FAILURES   100     /* Termination criterion. */
#define MAX_STEPS      500000  /* Stability criterion */
#define MAX_EXEC       1       /* Number of executions of the whole program */
#define FMAX           10.0f   /* Maximum force to use */
#define RUNS           10      /* number of times the program will run inside an */
                               /* execution cicle */
#define PI             3.14159265358979323846f

#define  ALPHAINIC     0.05f
#define  BETAINIC      0.000002f

CMathrl     mathrl;
CCartpole   cartpole;
CRLearning  rlearning;
CFuzzy      fuzzy;
CRgenerator gen;

float       gain_sel[MAX_EXEC][2];
int         output[10];
//--------------------------------------------------------------------------

void printTrialsDistribution( int* trials ){

   //init output vector
   for( int i = 0; i < 10; i++ ) output[i] = 0;
   for( int i = 0; i < RUNS; i++ ){
      if(0*MAX_FAILURES/50<trials[i]&&trials[i]<=1*MAX_FAILURES/50)
         output[0] = output[0]++;
      else if(1*MAX_FAILURES/50<trials[i]&&trials[i]<=2*MAX_FAILURES/50)
         output[1] = output[1]++;
      else if(2*MAX_FAILURES/50<trials[i]&&trials[i]<=3*MAX_FAILURES/50)
         output[2] = output[2]++;
      else if(3*MAX_FAILURES/50<trials[i]&&trials[i]<=4*MAX_FAILURES/50)
         output[3] = output[3]++;
      else if(4*MAX_FAILURES/50<trials[i]&&trials[i]<=5*MAX_FAILURES/50)
         output[4] = output[4]++;
      else if(5*MAX_FAILURES/50<trials[i]&&trials[i]<=6*MAX_FAILURES/50)
         output[5] = output[5]++;
      else if(6*MAX_FAILURES/50<trials[i]&&trials[i]<=7*MAX_FAILURES/50)
         output[6] = output[6]++;
      else if(7*MAX_FAILURES/50<trials[i]&&trials[i]<=8*MAX_FAILURES/50)
         output[7] = output[7]++;
      else if(8*MAX_FAILURES/50<trials[i]&&trials[i]<=9*MAX_FAILURES/50)
         output[8] = output[8]++;
      else if(9*MAX_FAILURES/50<trials[i])
```

```
            output[9] = output[9]++;
      }

      printf("Trials Distribution \n");
      for( int i = 0; i < 10; i++ ){
         printf( "%d ", output[i] );
      }
}

int _tmain( int argc, _TCHAR* argv[] ){
      float force;
      float N;
      float stoch_noise;
      float avg_trials;
      float alpha;
      float beta;
      float E     = 0.0;
      float temp  = 0.0;
      float alpha_rate    =       0.0f;   // Set to 0 for non changing config
      float beta_rate     =       1.0f;   // Set to 1 for non changing config
      int   steps = 0,
            exec  = 0,
            failures= 0,
            failed   = 0,
            trials[RUNS],
            run      = 0;

      alpha   (float)ALPHAINIC;
      beta  = (float)BETAINIC;

      while( exec < MAX_EXEC ){
         printf("Execution number: %d \n \n", exec + 1 );

         for( int i   0; i < RUNS; i++ ){
            trials[i] = 0;
         }

         /*--- Set centre and standard deviation values to Fuzzy MFs ---*/
         fuzzy.set_values( );

         run = 0;
         while( run < RUNS ){
            failures      = 0;
            steps         = 0;

            /*--- Initialize action and heuristic critic weights and traces ---*/
            rlearning.reset( );

            /*--- Starting state is (0 0 0 0) ---*/
            cartpole.reset( );

            /*--- Aquire inputs and fuzzyfy ---*/
            fuzzy.fuzzyfy( cartpole.get_x( ), cartpole.get_x_dot( ), cartpole.get_x_dotdot(
), cartpole.get_theta( ), cartpole.get_theta_dot( ), cartpole.get_theta_dotdot( ) );

            /*--- Transfer mu values from fuzzy class to RL class ---*/
            for( int i   0; i < 16; i++ ){
               rlearning.set_mu( i, fuzzy.get_mu( i ) );
            }

            /*--- Reinforcement is 0. Prediction of failure given by v weight. ---*/
            rlearning.set_r( 0.0 );

            /*--- Prediction of failure for current state ---*/
            rlearning.set_p( 0.0 );
            rlearning.calc_oldp( );
            rlearning.calc_p( );

            /*--- Iterate through the action-learn loop. ---*/
```

```
        while( steps < MAX_STEPS && failures < MAX_FAILURES ){ // Executes 1 trial
           /*--- Calculatin the stochastic noise---*/
           N = 1 / ( 1 + exp( 2 * rlearning.get_p() ) );

           stoch_noise    ( float )gen.randn_notrig( 0, N ) / 10;

           /*--- Calculating output from ASE ---*/
           rlearning.calc_output( stoch_noise );

           /*--- calculating force to use ---*/
           force = FMAX * rlearning.get_output( );

           /*--- Apply action to the simulated cart-pole ---*/
           cartpole.calculate_cart_pole( force ),

           /*--- Aquire inputs and fuzzyfy ---*/
           fuzzy.fuzzyfy( cartpole.get_x( ), cartpole.get_x_dot( ),
cartpole.get_x_dotdot( ), cartpole.get_theta( ), cartpole.get_theta_dot( ),
cartpole.get_theta_dotdot( ) );
           for( int i = 0; i < 16; i++ ){
              rlearning.set_mu( i, fuzzy.get_mu( i ) );
           }

           if( fuzzy.fail_flag ){
              /*--- Failure occurred. ---*/
              failed = 1;
              failures++;
              printf( "Trial %d was %d steps\n", failures, steps );
              steps = 0;

              /*--- Starting state is (0 0 0 0) ---*/
              cartpole.reset( );

              /*--- Aquire inputs and fuzzyfy ---*/
              fuzzy.fuzzyfy( cartpole.get_x( ),
                      cartpole.get_x_dot( ),
                      cartpole.get_x_dotdot( ),
                      cartpole.get_theta( ),
                      cartpole.get_theta_dot( ),
                      cartpole.get_theta_dotdot( ) );
              for( int i = 0; i < 16; i++ ){
                 rlearning.set_mu( i, fuzzy.get_mu( i ) );
              }

              /*--- Reinforcement upon failure is -1. Prediction of failure is 0. ---*/
              rlearning.set_r( -1.0 );
              rlearning.set_p( 0.0 );
              rlearning.calc_oldp( );
           }
           else{
              /*--- Not a failure. ---*/
              failed = 0;

              /*--- Reinforcement is 0. Prediction of failure given by v weight. ---*/
              rlearning.set_r( 0.0 ),
              rlearning.calc_oldp( ),
              rlearning.calc_p( );

           }

           /*--- Heuristic reinforcement is: current rein forcement + gamma * new
failure prediction   previous failure prediction ---*/
           rlearning.calc_rhat( );

           /*--- Calculating E ---*/
           E = (float)( 0.5 * rlearning.get_rhat( ) * rlearning.get_rhat( ) );

           /*--- Updating the learning weights ---*/
           rlearning.upd_weights( alpha, beta );
```

```
            if( failed ){
               /*--- If failure, zero all traces. ---*/
               rlearning.reset_traces( );
            }
            else{
               /*--- Otherwise, update (decay) the traces. ---*/
               rlearning.upd_traces( );
            }
            steps++;
         }
         if (failures >= MAX_FAILURES){
            printf("Pole not balanced. Stopping after %d failures \n \n", failures );
            trials[run] = (int) failures;
         }
         else{
            trials[run] = (int) failures + 1;
            printf("Pole balanced successfully for at least %d steps \n \n", steps );
         }
         if( run == RUNS   1 ){
            for( int i = 0; i < MAX_STEPS; i++ ){
            }
         }
         run++;
      }

      temp = 0;
      for( int i = 0; i < RUNS; i++ ){
         temp += (float)trials[i]   1.0f;
         }
      avg_trials = (float)(temp / RUNS);
      printf( "Average number of trials was %f \n \n", avg_trials );

      printTrialsDistribution(trials);
      printf("\n\n" ),

      avg_trials=(float)((temp-(MAX_FAILURES*output[9]))/(RUNS-output[9]));
      printf("Real average number of trials was %f \n \n \n", avg_trials );

      gain_sel[exec][0]   avg_trials;
      gain_sel[exec][1] = (float)output[9];
      exec++;

      alpha += alpha_rate;
      beta  *= beta_rate;
   }
   printf("Alpha distribution: \n");
   for( int i = 0; i < MAX_EXEC; i++ ){
      printf("%f  ", gain_sel[i][0]),
   }
   printf("\n");
   for( int i = 0; i < MAX_EXEC; i++ ){
      printf("%f  ", gain_sel[i][1]);
   }

   while( 1 );
   return 0;
}
```

REINFORCEMENT LEARNING PROGRAM

```
#include "StdAfx.h"
#include "RLearning.h"
#include "math.h"

CRLearning::CRLearning( void ) { }
CRLearning::~CRLearning( void ){ }

void CRLearning::reset( void ){
   for( int i = 0; i < NSTATES; i++ ){
      C_w[i]       0.0;
      C_v[i]     = 0.0;
      C_muhat[i] = 0.0;
      C_e[i]     = 0.0;
   }
}
//---------------------------------------------------------------------------

void CRLearning::reset_traces( void ){
   for( int i = 0; i < NSTATES; i++ ){
      C_e[i]     = 0;
      C_muhat[i] = 0;
   }
}
//---------------------------------------------------------------------------

void CRLearning::set_r( float r ){
   C_r = r;
}
//---------------------------------------------------------------------------

void CRLearning::set_p( float p ){
   C_p = p;
}
//---------------------------------------------------------------------------

void CRLearning::set_mu( int k, float mu ){
   C_mu[k] = mu;
}
//---------------------------------------------------------------------------

/* Calculates the final probability of executing an action */
void CRLearning::calc_output( float noise ){
   float temp1 = 0;
   float temp2 = 0;

   for( int i = 0; i < NSTATES; i++ ){
      C_ro[i] = sfunc( C_w[i] );
      temp1 += C_ro[i] * C_mu[i];
      temp2 += C_mu[i];
   }

   C_P      = temp1/temp2;
   C_output = C_P + noise;
}
//---------------------------------------------------------------------------
```

```
/* Calculates the internal reinforcement signal */
void CRLearning::calc_rhat( void ){
   C_rhat = C_r + ( GAMMA * C_p )   C_oldp;
}
//---------------------------------------------------------------------------

/* Calculates the prediction of eventual reinforcement */
void CRLearning::calc_p( void ){
   C_p = 0;
   for( int i = 0; i < NSTATES; i++)
      C_p += ( C_v[i] * C_mu[i] );
}
//---------------------------------------------------------------------------

/* Stores old value of p */
void CRLearning::calc_oldp( void ){
   C_oldp   C_p;
}
//---------------------------------------------------------------------------

/* Update the weights of v and w */
void CRLearning::upd_weights( float A, float B ){
   for( int i = 0; i < NSTATES; i++){
      C_w[i] += A * C_rhat * C_mu[i] * C_e[i];
      C_v[i] -= B * C_rhat * C_mu[i];
      if( C_w[i] >= 10 ) C_w[i] = 10;
      else if( C_w[i] <= -10 ) C_w[i] = -10;
   }
}
//---------------------------------------------------------------------------

/* Update the value of the traces */
void CRLearning::upd_traces( void ){
   for( int i = 0; i < NSTATES; i++ ){
      C_e[i] = (1 / C_ro[i]) * exp(-C_w[i]) ;
   }
}
//---------------------------------------------------------------------------

float CRLearning::TSugeno1( void ){
   float temp1 = 0.0;
   float temp2   0.0;

   for( int i = 0; i < NSTATES; i++ ){
      temp1 += ( C_am[i] * C_mu[i] );
      temp2 += C_mu[i];
   }
   return( temp1 / temp2) ;
}
//---------------------------------------------------------------------------

float CRLearning::TSugeno2( void ){
   float temp1 = 0.0;
   float temp2 = 0.0;

   for( int i = 0; i < NSTATES; i++ ){
      temp1 += ( C_ro[i] * C_mu[i] );
      temp2 += C_mu[i];
   }
   return( temp1 / temp2 );
}
//---------------------------------------------------------------------------

/* Implements an S-shaped function */
float CRLearning::sfunc( float s ){
   return ( ( float )( 2.0 / ( 1.0 + exp( -s ) ) )   1 );
}
//---------------------------------------------------------------------------
```

```
/* Implements a selecting function */
float CRLearning::gfunc( float s ){
   if( s >= 0.5 ) return 1.0;
   else  return -1.0;
}
//---------------------------------------------------------------------------

float CRLearning::get_rhat( void ){
   return C_rhat;
}
//---------------------------------------------------------------------------

float CRLearning::get_output( void ){
   return C_output;
}
//---------------------------------------------------------------------------

float CRLearning::get_p( void ){
   return C_p;
}
//---------------------------------------------------------------------------
```

FUZZY LOGIC PROGRAM

```
#include "StdAfx.h"
#include "Fuzzy.h"
#include <math.h>

CFuzzy::CFuzzy(void){}
CFuzzy::~CFuzzy(void){}
//---------------------------------------------------------------------------

void CFuzzy::set_values(void){
   set_centers();
   set_sigmas();
}
//---------------------------------------------------------------------------

void CFuzzy::set_centers( void ){
   // x
   input1.centre[0] = -1.5f;
   input1.centre[1] = 1.5f;

   // x_dot
   input2.centre[0] = -0.5f;
   input2.centre[1] = 0.5f;

   // theta
   input3.centre[0] = -1.8f;
   input3.centre[1] = 1.8f;

   // theta_dot
   input4.centre[0] = -0.05f;
   input4.centre[1] = 0.05f;
}
//---------------------------------------------------------------------------

float CFuzzy::gaussian( float x, float c, float s ){
   float tmp1, tmp2;

   tmp1 = ( x - c ) / s;
   tmp2 = ( float )( ( -0.5 ) * tmp1 * tmp1 );

   return exp( tmp2 );
}
//---------------------------------------------------------------------------

float CFuzzy::gaussian2( float x, float c, float s ){
   float tmp1, tmp2;

   if( c < 0 ){
      if( x <= c ) return 1;
      else{
         tmp1 = ( x   c ) / s;
         tmp2 = ( float )( ( -0.5 ) * tmp1 * tmp1 );
         return exp( tmp2 );
      }
   }
   else{
      if( x >= c ) return 1;
```

```
      else{
         tmp1 = ( x    c ) / s;
         tmp2 = ( float )( -0.5 * tmp1 * tmp1 );
         return exp( tmp2 );
      }
   }
}
//---------------------------------------------------------------------------

void CFuzzy::fuzzyfy(float x1, float x2, float x3, float x4, float x5, float x6 ){
   int cnt1, cnt2, cnt3, cnt4;
   int i   0;

   C_c1 = x1;
   C_c2 = x2;
   C_c3   x3;
   C_c4   x4;
   C_c5   x5;
   C_c6   x6;

   for( cnt1 = 0; cnt1 <2; cnt1++ ){
      for( cnt2 = 0; cnt2 <2; cnt2++ ){
         for( cnt3 = 0; cnt3 <2; cnt3++ ){
            for( cnt4 = 0; cnt4 < 2; cnt4++ ){
               C_u1 = gaussian2( C_c1, input1.centre[cnt1], input1.sigma[cnt1] );
               C_u2 = gaussian2( C_c2, input2.centre[cnt2], input2.sigma[cnt2] );
               C_u3 = gaussian2( C_c4, input3.centre[cnt3], input3.sigma[cnt3] );
               C_u4 = gaussian2( C_c5, input4.centre[cnt4], input4.sigma[cnt4] );

               mu[i] = MIN( C_u1, C_u2, C_u3, C_u4 );
               if( mu[i] <= 0.000001f ) mu[i] = 0.000001f;
               i++;
            }
         }
      }
   }

   /*--- To signal failure ---*/
   if( C_c1 < -2.4 || C_c1 > 2.4 || C_c4 < MINANGLE || C_c4 > MAXANGLE )
      fail_flag = 1;   // Failure
   else fail_flag = 0; // Not failure
}
//---------------------------------------------------------------------------

float CFuzzy::get_mu( int k ){
   return mu[k];
}
//---------------------------------------------------------------------------

float CFuzzy::get_centre( int in, int c ){
   switch ( in ){
      case 1:  return input1.centre[c];
      case 2:  return input2.centre[c];
      case 3:  return input3.centre[c];
      case 4:  return input4.centre[c];
   }
}
//---------------------------------------------------------------------------

float CFuzzy::MIN( float a, float b, float c, float d ){
   float temp;

   temp = a;
   if( temp >= b ) temp = b;
   if( temp >= c ) temp = c;
   if( temp >= d ) temp = d;

   return temp;
}
```

```
//---------------------------------------------------------------------------

void CFuzzy::set_sigmas( void ){

   // x
   input1.sigma[0] = input1.centre[1] * SIGMA;
   input1.sigma[1]   input1.centre[1] * SIGMA;

   // x_dot
   input2.sigma[0]   input2.centre[1] * SIGMA;
   input2.sigma[1]   input2.centre[1] * SIGMA;

   // theta
   input3.sigma[0]   input3.centre[1] * SIGMA;
   input3.sigma[1] = input3.centre[1] * SIGMA;

   // theta_dot
   input4.sigma[0] = input4.centre[1] * SIGMA;
   input4.sigma[1] = input4.centre[1] * SIGMA;
}
//---------------------------------------------------------------------------

void CFuzzy::adapt_sigmas( ){
}
//---------------------------------------------------------------------------
```

CART-POLE MODEL

```
#include "StdAfx.h"
#include "Cartpole.h"
#include <string>
#include <iostream>
#include "math.h"

#define GRAVITY               -9.8f
#define MASSPOLE              0.1f
#define MASSCART              1.0f
#define LENGTH                0.5f
#define FRICTION_COEF_CART    0.0005f
#define FRICTION_COEF_POLE    0.000002f

#define TOTALMASS ( MASSPOLE + MASSCART )
#define MASSPOLE_LENGTH ( MASSPOLE * LENGTH )

#define FOURTHIRDS            1.33333333333333333f

#define TAU                   0.02f /*seconds between state updates*/

#define M_PI                  3.14159265358979323846f

CCartpole::CCartpole( ){
   this->theta = 0;
   this->theta_dot = 0;
   this->theta_dotdot = 0;
   this->x = 0;
   this->x_dot = 0;
   this->x_dotdot = 0;
   this->Nc = 1; //It has to be positive in the first loop
   this->frictionf = ( FRICTION_COEF_CART * sgn( this->Nc * this->x_dot ) );
}

CCartpole::~CCartpole( void ){}

/*--- Implementation of the dinamics of the cart-pole system ---*/
void CCartpole::calculate_cart_pole( float force ){
   float numerator, costheta, sintheta, brackets, denominator, theta_dot_square,
denominator_brackets;

   costheta = cos(this->theta);
   sintheta = sin(this->theta);
   theta_dot_square = this->theta_dot * this->theta_dot;

   /*--- Calculate theta_dot_dot ---*/
   brackets=( force + ( MASSPOLE_LENGTH * theta_dot_square * sintheta )   ( sgn( this-
>x_dot ) * FRICTION_COEF_CART ) ) / TOTALMASS;

   numerator = ( GRAVITY * sintheta ) - costheta * brackets   ( ( FRICTION_COEF_POLE *
this->theta_dot ) / ( MASSPOLE_LENGTH ) );

   denominator_brackets = FOURTHIRDS   ( ( MASSPOLE * costheta * costheta ) / TOTALMASS
);

   denominator = LENGTH * denominator_brackets;
   this->theta_dotdot = numerator / denominator;
```

```
   /*--- Calculate x_dotdot ---*/
   this->x_dotdot    ( force + ( MASSPOLE_LENGTH * ( ( theta_dot_square * sintheta ) - (
this->theta_dotdot * costheta ) ) )    ( sgn( this->x_dot ) * FRICTION_COEF_CART ) ) /
TOTALMASS;

   /*--- Update state variables (Euler's method) ---*/
   this->x             += TAU * this->x_dot;
   this->x_dot         += TAU * this->x_dotdot;
   this->theta         += TAU * this->theta_dot;
   this->theta_dot     += TAU * this->theta_dotdot;
}

void CCartpole::reset( ){
   this->theta = 0;
   this->theta_dot   0;
   this->theta_dotdot = 0;
   this->x = 0;
   this->x_dot = 0;
   this->x_dotdot   0;
   this->Nc = 1;
   this->frictionf = (float)(FRICTION_COEF_CART * sgn(this->Nc * this->x_dot));
}

//Consultant functions
float CCartpole::get_theta( ){
   return this->theta;
}

float CCartpole::get_theta_dot( ){
   return this->theta_dot;
}

float CCartpole::get_theta_dotdot( ){
   return this->theta_dotdot;
}

float CCartpole::get_x( ){
   return this->x;
}

float CCartpole::get_x_dot( ){
   return this->x_dot;
}

float CCartpole::get_x_dotdot( ){
   return this->x_dotdot;
}

//Math functions
float CCartpole::sgn( float x ){
   if( x < 0 ) return -1;
   else if( x >= 0 ) return 1;
}

/* To calculate sin of a number in degrees */
float CCartpole::sin2( float degrees ){
   return sin( degrees * M_PI / 180 );
}

/* To calculate cosine of a number in degrees */
float CCartpole::cos2( float degrees ){
   return cos( degrees * M_PI / 180 );
}
```

RANDOM NOISE GENERATOR

```
#include "StdAfx.h"
#include "Rgenerator.h"
#include <cmath>
#include <cstdlib>

CRgenerator::CRgenerator(void){}
CRgenerator::~CRgenerator(void){}

//"Polar" version without trigonometric calls
double CRgenerator::randn_notrig(double mu, double sigma) {

   static bool deviateAvailable=false;       // flag
   static float storedDeviate;               // deviate from previous calculation
   double polar, rsquared, var1, var2;

   // If no deviate has been stored, the polar Box-Muller transformation is
   // performed, producing two independent normally-distributed random
   // deviates. One is stored for the next round, and one is returned.
   if (!deviateAvailable) {

      // Choose pairs of uniformly distributed deviates, discarding those
      // that don't fall within the unit circle
      do {
         var1=2.0*( double(rand())/double(RAND_MAX) )   1.0;
         var2=2.0*( double(rand())/double(RAND_MAX) )   1.0;
         rsquared=var1*var1+var2*var2;
      } while ( rsquared>=1.0 || rsquared == 0.0);

      // Calculate polar tranformation for each deviate
      polar=sqrt(-2.0*log(rsquared)/rsquared);

      // Store first deviate and set flag
      storedDeviate=var1*polar;
      deviateAvailable=true;

      // Return second deviate
      return var2*polar*sigma + mu;
   }

   // If a deviate is available from a previous call to this function, it is
   // returned, and the flag is set to false.
   else {
      deviateAvailable=false;
      return storedDeviate*sigma + mu;
   }
}

/*****************************************************************************/
// Standard version with trigonometric calls
#define PI 3.14159265358979323846

double CRgenerator::randn_trig(double mu, double sigma) {
   static bool deviateAvailable=false;       // Flag
   static float storedDeviate;               // Deviate from previous calculation
   double dist, angle;
```

```
    // If no deviate has been stored, the standard Box-Muller transformation is
    // performed, producing two independent normally-distributed random
    // deviates. One is stored for the next round, and one is returned.
    if (!deviateAvailable) {

        // Choose a pair of uniformly distributed deviates, one for the
        // distance and one for the angle, and perform transformations
        dist=sqrt( -2.0 * log(double(rand()) / double(RAND_MAX)) );
        angle=2.0 * PI * (double(rand()) / double(RAND_MAX));

        // Calculate and store first deviate and set flag
        storedDeviate=dist*cos(angle);
        deviateAvailable=true;

        // Calculate return second deviate
        return dist * sin(angle) * sigma + mu;
    }

    // If a deviate is available from a previous call to this function, it is
    // returned, and the flag is set to false.
    else {
        deviateAvailable=false;
        return storedDeviate*sigma + mu;
    }
}
```

MATHEMATICAL FUNCTIONS

```
#include "StdAfx.h"
#include "Mathrl.h"
#include <math.h>

CMathrl::CMathrl(void){}
CMathrl::~CMathrl(void){}
//-----------------------------------------------------------------------------

// Returns the result of multiplication operation
float CMathrl::cmul( float C_op_A, float C_op_B ){
   return( C_op_A * C_op_B );
}
//-----------------------------------------------------------------------------

// Returns the logic "and"
float CMathrl::cand( float C_op_A, float C_op_B ){
   if( C_op_A > C_op_B )      return( C_op_B );
   else  return( C_op_A );
}
//-----------------------------------------------------------------------------

// Returns the minimum of the set
float CMathrl::cmin( float C_op_A, float C_op_B ){
   if( C_op_A > C_op_B )      return( C_op_B );
   else  return( C_op_A );
}
//-----------------------------------------------------------------------------

// Returns the maximum of the setfloat max(float x, float y);
float CMathrl::cmax( float C_op_A, float C_op_B ){
   if( C_op_A > C_op_A )      return( C_op_A );
   else  return( C_op_B );
}
//-----------------------------------------------------------------------------

// Returns the s-shaped function to limit between -1 and 1
float CMathrl::csfunc( float C_s ){
   float temp;
   temp = (float)( 1.0 / ( 1.0 + exp( C_s ) ) );
   return( temp );
}
//-----------------------------------------------------------------------------
```

Appendix C

DC Motor Control Problem

DC MOTOR MODEL

```
#pragma once
#include "Actuators.h"
//---------------------------------------------------------------------------

/*--- Inicializing parameters ---*/
void DCMotor::init(void){
   A_theta_dot = 0.0;
   A_i         = 0.0;
   A_i_dot     = 0.0;
   A_theta     = 0.0;
}
//---------------------------------------------------------------------------

/*--- DC Motor model ---*/
void DCMotor::run( float A_volt, float A_torque){
   float A_thetaacc, A_temp1, A_temp2;

   A_temp1  = A_volt - ( Res * A_i )   ( Ke * A_theta_dot );
   A_i_dot  = A_temp1 / L;

   A_temp2     = ( Kt * A_i ) - (Damp * A_theta_dot)   A_torque;
   A_thetaacc  = A_temp2 / J;

   // Update the four state variables, using Euler's method.
   A_theta     += TAU * A_theta_dot;
   A_theta_dot += TAU * A_thetaacc;
   A_i         += TAU * A_i_dot;
}
//---------------------------------------------------------------------------
```

MOTOR-MASS SYSTEM

```
#pragma once
#include "System.h"
//-------------------------------------------------------------------------------

/*--- Mass for motor ---*/
float motor_mass( float S_theta ){
   float S_sintheta, S_torque;

   S_sintheta  = sin(S_theta);
   S_torque    = M * G * d * S_sintheta;

   return S_torque;
}
//-------------------------------------------------------------------------------

/*--- Sign function ---*/
float sgn (float value){
   if ( value <= 0 )   return -1;
   else  return 1;
}
//-------------------------------------------------------------------------------
```

REINFORCEMENT LEARNING PROGRAM

```
#pragma once
#include "RLearning.h"
//---------------------------------------------------------------------------

void CRLearning::init_rl_values( void ){
   for (int i = 0; i < N_RULES; i++) {
      Crl_w[i]         = 0.0;
      Crl_v[i]         = 0.0;
      Crl_ubar[i]      = 0.0;
      Crl_e[i]         = 0.0;
      Crl_prob[i]      = random( 100 );
   }
}
//---------------------------------------------------------------------------

void CRLearning::set_state(float u, int k){
   Crl_u[k] = u;
}
//---------------------------------------------------------------------------

/*--- Computes the probability of success ---*/
void CRLearning::get_prob( ){
   float temp1 = 0.0;
   float temp2 = 0.0;

   for (int i = 0; i < N_RULES; i++) {
      Crl_prob[i] = s_shaped( Crl_w[i] );
      Crl_ace[i]  = s_shaped( Crl_v[i] );
      temp1       += Crl_prob[i] * Crl_u[i];
      temp2       += Crl_u[i];
   }

   Crl_y = temp1 / temp2;
}
//---------------------------------------------------------------------------

/*--- Choose an action ---*/
void CRLearning::get_action( ){
   if( 0.5 < Crl_y )   Crl_a = 1;
   else  Crl_a   -1;
}
//---------------------------------------------------------------------------

void CRLearning::upd_trace( ){
    for (int I    = 0; i < N_RULES; i++) {
      Crl_e[i]    = (1 / Crl_prob[i] ) * exp(-Crl_w[i]);
    }
}
//---------------------------------------------------------------------------

/*--- Computes the future failure prediction ---*/
void CRLearning::get_p( ){
    // Remember prediction of failure for current state.
    Crl_oldp = Crl_p;
    Crl_p = 0;
```

```
    if (!Crl_fail) {
      // Computing the future failure prediction
      for (int i = 0; i < N_RULES; i++) {
         Crl_p += Crl_v[i] * Crl_u[i];
      }
    }
}
//--------------------------------------------------------------------------

/*--- Updating the weights ---*/
void CRLearning::upd_weights( ){
   // Computing Heuristic reinforcement
   Crl_rhat = (float)Crl_r + GAMMA * ( Crl_p   Crl_oldp);

   for (int i = 0; i < N_RULES; i++) {
      // Update all weights.
      Crl_w[i] += ALPHA * Crl_rhat * Crl_u[i] * Crl_e[i];
      Crl_v[i] -= BETA * Crl_rhat * Crl_u[i];
   }
}
//--------------------------------------------------------------------------

void CRLearning::upd_r(int r, int error){
   Crl_r = r;
   Crl_error = error;
   if( Crl_error >= 0 )       Crl_error = -1;
   else  Crl_error   1;
}
//--------------------------------------------------------------------------

/*--- Set the random range ---*/
void CRLearning::set_random_range(float range){
   Crl_rdm_range = range;
}
//--------------------------------------------------------------------------

/*--- Set the fail flag ---*/
void CRLearning::set_fail_flag(bool fail){
   Crl_fail = fail;
}
//--------------------------------------------------------------------------

void CRLearning::reset_traces( ){
   for (int i = 0; i < N_RULES; i++) {
      Crl_e[i]    = 0.0;
   }
}
```

FUZZY LOGIC PROGRAM

```
#pragma once
#include "Fuzzy.h"
//---------------------------------------------------------------------------

/*--- Setting the centres of the input membership functions ---*/
void CFuzzy::set_centers( ){
   // Error MF
   CF_MF_J1[0] = -85;
   CF_MF_J1[1] = -52;
   CF_MF_J1[2] = -16.5;
   CF_MF_J1[3] = 16.5;
   CF_MF_J1[4] = 52;
   CF_MF_J1[5] = 85;

   // Error rate of change MF
   CF_MF_J2[0]   -70;
   CF_MF_J2[1] = -35;
   CF_MF_J2[2] = 0;
   CF_MF_J2[3] = 35;
   CF_MF_J2[4] = 70;
}
//---------------------------------------------------------------------------

/*--- Setting the standard deviations of the input membership functions ---*/
void CFuzzy::set_sigmas( ){
   // Error MF
   CF_MF_SJ1[0] = 18;
   CF_MF_SJ1[1] = 14;
   CF_MF_SJ1[2] = 5;
   CF_MF_SJ1[3] = 5;
   CF_MF_SJ1[4] = 14;
   CF_MF_SJ1[5] = 18;

   // Error rate of change MF
   CF_MF_SJ2[0]   15;
   CF_MF_SJ2[1]   15;
   CF_MF_SJ2[2]   15;
   CF_MF_SJ2[3]   15;
   CF_MF_SJ2[4] = 15;
}
//---------------------------------------------------------------------------

void CFuzzy::set_sigma( float sig ){
   CF_SIGMA = sig;
}
//---------------------------------------------------------------------------

/*--- Gaussian function ---*/
float CFuzzy::Gaussian(float x, float c){
   float tmp1, tmp2;
   tmp1 = ( x   c ) / CF_SIGMA;
   tmp2 = -0.5 * tmp1 * tmp1;
   return exp( tmp2 );
}
//---------------------------------------------------------------------------
```

```
/*--- Fuzzyfication ---*/
void CFuzzy::Fuzzy_G(float X1, float X2){
   int k = 0;
   float temp1, temp2, tempc, temps;

   for (int j = 0; j < 5; j++) { // Error dot
      tempc = CF_MF_J2[j];
      temps = CF_MF_SJ2[j];
      set_sigma(temps);
      temp2 = Gaussian(X2, tempc );

      // Error
      for( int i = 0; i < 6; i++ ){
         tempc = CF_MF_J1[i];
         temps = CF_MF_SJ1[i];
         set_sigma(temps);
         temp1 = Gaussian( X1, tempc );
         if ( i == 0 ) {
            if (X1 <= CF_MF_J1[0]) {
               temp1 = 1;
            }
         }
         if ( i == 5 ) {
            if (X1 >= CF_MF_J1[5]) {
               temp1 = 1;
            }
         }
         CF_u[k] = MUL( temp1, temp2 );
         k++;
      }
   }
}
```

MATHEMATICAL FUNCTIONS

```
#include "mFunctions.h"

using namespace std;

#ifdef _DEBUG
#define new DEBUG_NEW
#endif
//---------------------------------------------------------------------------

/*--- Returns the minimum of the set ---*/
float min( float mF_x, float mF_y ){
   if( mF_x > mF_y )   return( mF_y );
   else  return( mF_x );
}
//---------------------------------------------------------------------------

/*--- Returns the maximum of the setfloat ---*/
float max( float mF_x, float mF_y ){
   if( mF_x > mF_y )   return( mF_x );
   else  return( mF_y );
}
//---------------------------------------------------------------------------

/*--- Calculates the probability of succes if turning right, from 0 to 100% ---*/
float s_shaped( float mF_s ){
   float mF_x;

   mF_x = 1 / ( 1.0 + exp( -max( -50.0, min( mF_s, 50.0 ) ) ) );

   return mF_x;
}
//---------------------------------------------------------------------------

/*--- Make random floating point numbers in interval from 0 to 1 ---*/
float random( float mF_range ){
   double   mF_r;
   int32 mF_seed;

   // random seed
   mF_seed   time( 0 );

   // Uses the seed to generate a Mersenne random number
   TRandomMersenne rg( mF_seed );

   // Random number will be -0.5 to 0.5
   mF_r = ( mF_range * ( rg.Random( )   0.5 ) ) / 100;

   return ( ( float ) mF_r );
}
//---------------------------------------------------------------------------

float MUL (float op_A, float op_B){
   return( op_A * op_B );
}
//---------------------------------------------------------------------------
```

```
float AND (float op_A, float op_B){
   if( op_A > op_B ) return( op_B );
   return( op_A );
}
```

ITAE/IAE CALCULATION

```
#pragma once
#include "Error.h"
//-------------------------------------------------------------------------------

void CError::init(){
   CE_i           = 0;
   CE_error       = 0.0;
   CE_error_old   = 0.0;
   CE_error_dot   = 0.0;
   first_run      = true;
   CE_fail        = false;
}
//-------------------------------------------------------------------------------

void CError::set_theta_ref(float theta_ref){
   CE_theta_ref     theta_ref;
}
//-------------------------------------------------------------------------------

void CError::set_perf_limit(float itae_limit, float iae_limit){
   CE_ITAE_limit  = itae_limit;
   CE_IAE_limit   = iae_limit;
}
//-------------------------------------------------------------------------------

void CError::reset_perf(){
   CE_ITAE  = 0.0;
   CE_IAE   = 0.0;
}
//-------------------------------------------------------------------------------

void CError::get_error(float theta, float t){
   float error_dot_tmp;

   CE_theta_old   = CE_theta;
   CE_theta       = theta;
   CE_theta_dot   = CE_theta   CE_theta_old;

   CE_error_old   = CE_error;
   CE_error       = CE_theta_ref   CE_theta;
   error_dot_tmp  = CE_error - CE_error_old;

   if (error_dot_tmp > 0.1)    error_dot_tmp    0.1;
   if (error_dot_tmp < -0.1)   error_dot_tmp    -0.1;

   CE_error_dot   - 10000 * error_dot_tmp;

   // Failure detection
   CE_ITAE += t * (abs(CE_error)) * TAU;

   if (CE_ITAE <= 0) CE_ITAE = 0;

   if (t < 0) ;;

   CE_IAE += (abs(CE_error)) * TAU;
```

```
    if (CE_ITAE < CE_ITAE_limit & CE_IAE < CE_IAE_limit) {
       CE_r     = 0;
       CE_fail  = false;
    }
    else {
       CE_r     = -1;
       CE_fail  = true;
    }

    if (first_run) {
       CE_error_dot       0.0;
       first_run          false;
    }
}
```

Appendix D

Khepera III – Obstacle Avoidance Problem

MAIN

```
// Mthreading_Serial.cpp . Defines the entry point for the console application.

#include "stdafx.h"
#include "Serial.h"
#include <iostream>
#include <stdlib.h>
#include <windows.h>
#include <process.h>   // needed for _beginthread()

using namespace std;
//---------------------------------------------------------------------------

// Constants
#define SENSORNUMBER   13      // Total number of sensor
#define  AGENTNUM      4       // Number of learning agents
#define RANGE          4000    // Range of sensors
#define  MAXPROX       0.15    // Minimum distance before being considered a failure
#define CENTREF        0.20    // Centre of far gaussian membership function
#define CENTREN        0.40    // Centre of near gaussian membership function
#define SIGMA          0.08    // Standard deviation for the gaussian functions
#define ALPHA          7       // Actor learning rate
#define BETA           0.03    // Critic learning rate
#define GAMMA          0.95    // Discounted temporal difference constant
#define SPEEDMAX       12000   // Maximum allowed speed
//---------------------------------------------------------------------------

// Function prototypes
void Initialize( void );
void PrintMatrix( char select );

// Threads
void KheperaCom (void);

// Serial
int OpenSerial(int com);
int SerialWrite( int command );
int SerialRead( void );
int SerialClose( void );
int ShowError (LONG lError, LPCTSTR lptszMessage);

// Probabilistic Fuzzy
void Fuzzyfi( void );
float Min( float op_A, float op_B, float op_C, float op_D );
float Max( float op_A, float op_B );
float Mul( float op_A, float op_B, float op_C, float op_D );
float Gauss_far( float x, float c, float s );
float Gauss_near( float x, float c, float s );
float s_shape( float s );

// Reinforcement Learning
void Reinforce( void );
//---------------------------------------------------------------------------

// Variables
int      LeftSpeed;    // Left wheel speed
int      RightSpeed;   // Right wheel speed
```

```
int      sensnum       = 0;
int      converted     = 0;
int      Sensor[SENSORNUMBER];        // IR sensor data is stored here
float X[SENSORNUMBER];                // Normalized sensors data
float XCluster[4];                    // Clustered sensors data
float mu[16];                         // System state matrix
float p[AGENTNUM];
float p_old[AGENTNUM];
float old_ro[AGENTNUM][SENSORNUMBER];
float ro[AGENTNUM][SENSORNUMBER];
float w[AGENTNUM][SENSORNUMBER];
float v[AGENTNUM][SENSORNUMBER];
float E[AGENTNUM];
float P[AGENTNUM];
float temp[2];
float r_hat[AGENTNUM];
float r[AGENTNUM];
int      behaviour[AGENTNUM][2] = {   { 2.5, 2.5},
                                      {-1, -1},
                                      {-1, 1},
                                      { 1, -1}
                                      };
char  LSpdStr[5];
char  RSpdStr[5];
char  Data[101];
char  szBuffer[101];
int   COM = 1;
LONG  lLastError = ERROR_SUCCESS;
DWORD dwBytesRead   0;
enum{    EOF_Char = 13 };
//---------------------------------------------------------------------------

// Clasess declarations
CSerial serial;

//---------------------------------------------------------------------------
// Main thread
int _tmain( int argc, _TCHAR* argv[] ){

   // Initializing data
   Initialize();

   // Opening serial port
   OpenSerial( COM );

   // Waiting loop
   while( true ){
      KheperaCom();
      Fuzzyfi();
      Reinforce(),
   }

   // Closing serial port
   SerialClose();

   // Exiting main function
   return 0;
}

//---------------------------------------------------------------------------
// Thread to send and receive data from Khepera sensors and to Khepera motors
void KheperaCom (void){
      SerialWrite( 4 );       // Send motors speeds
      Sleep(100);             // Gives time to the serial port to pass the data
      SerialWrite( 3 );       // Send request of sensor data
      SerialRead();           // Read data sent from Khepera Robot and store it in
                              // the global variable Sensor[]
}
```

```
//---------------------------------------------------------------------------
// Initialize robot data
void Initialize( void ){
   for( int j = 0; j < SENSORNUMBER; j++){
      Sensor[j] = 0;
   }

   for( int i   0; i < 4; i++){
      p[i]  = 0;
      p_old[i]= 0;
   }

   for( int i = 0; i < 4; i++){
      for( int j = 0; j < SENSORNUMBER; j++ ){
         v[i][j] = 0;          // Critic weight initialization
         w[i][j] = 0;          // Actor weight initialization
      }
   }
}

////////////////////////////////////////////////////////////////////////////
// Serial functions

//---------------------------------------------------------------------------
// Opens the serial port
int OpenSerial( int com ){

   int flag = 1;

   // Attempt to open the serial port (COM1)
   lLastError = serial.Open(_T("COM1"),0,0,false);
   if (lLastError != ERROR_SUCCESS)
      return ::ShowError(serial.GetLastError(), _T("Unable to open COM-port"));
   else{
      printf("COM successfully open\n");
      flag   0;
   }

   // Setup the serial port (115200,8,N,1) using hardware handshaking
   lLastError = serial.Setup( CSerial::EBaud115200,
                              CSerial::EData8,
                              CSerial::EParNone,
                              CSerial::EStop1 );
   if( lLastError != ERROR_SUCCESS )
      return ::ShowError( serial.GetLastError( ), _T("Unable to set COM-port setting" )
);
   else{
      printf( "Setup done\n" );
      flag = 0;
   }

   // Setup handshaking
   lLastError = serial.SetupHandshaking( CSerial::EHandshakeOff );
   if( lLastError != ERROR_SUCCESS )
      return ::ShowError( serial.GetLastError( ), _T( "Unable to set COM-port
handshaking" ) );
   else{
      printf( "Handshaking done\n\n" );
      flag = 0;
   }

   return flag;
}

//---------------------------------------------------------------------------
// Sends data to Robot
int SerialWrite( int command ){

   char s_data[25];
```

```
    switch( command ){
    case 0:
       // Starts Braitenberg in IR mode
       // \r Adds the "return key pressed" command
       strcpy_s(s_data, "A,0\r");
       break;
    case 1:
       // Starts Braitenberg in Ultra Sound mode
       strcpy_s(s_data, "A,1\r");
       break;
    case 2:
       // Stops Braitenberg modes
       strcpy_s(s_data, "A,2\r");
       break;
    case 3
       // Requests IR sensors data
       strcpy_s(s_data, "N\r");
       break;
    case 4 :
       // Convert int to string
       converted = sprintf_s( s_data, "D,1%i,1%i\r", LeftSpeed, RightSpeed);
       break;
    }

    lLastError = serial.Write( s_data );
    if( lLastError != ERROR_SUCCESS )
       return ::ShowError(serial.GetLastError(), _T("Unable to send data"));

    return 0;
}

//---------------------------------------------------------------------------
// Reads data sent by robot
int SerialRead( void ){

    // Register only for the receive event
   lLastError = serial.SetMask(CSerial::EEventBreak |
                               CSerial::EEventCTS   |
                               CSerial::EEventDSR   |
                               CSerial::EEventError |
                               CSerial::EEventRing  |
                               CSerial::EEventRLSD  |
                               CSerial::EEventRecv);
    if (lLastError != ERROR_SUCCESS)
       return ::ShowError(serial.GetLastError(), _T("Unable to set COM-port event
mask"));

    // Use 'non-blocking' reads, because we don't know how many bytes
    // will be received. This is normally the most convenient mode
    // (and also the default mode for reading data).
    lLastError = serial.SetupReadTimeouts(CSerial::EReadTimeoutNonblocking);
    if (lLastError != ERROR_SUCCESS)
       return ::ShowError(serial.GetLastError(), _T("Unable to set COM-port read
timeout."));

    // Keep reading data, until an EOF (CTRL-Z) has been received
    int    j   0;
    bool fContinue  true;
    sensnum = 0;
    do{
       // Wait for an event
       lLastError = serial.WaitEvent();
       if (lLastError != ERROR_SUCCESS)
          return ::ShowError(serial.GetLastError(), _T("Unable to wait for a COM-port
event."));

       // Save event
       const CSerial::EEvent eEvent = serial.GetEventType();
```

```
// Handle break event
if (eEvent & CSerial::EEventBreak){
   printf("\n### BREAK received ###\n");
}

// Handle CTS event
if (eEvent & CSerial::EEventCTS){
   printf("\n### Clear to send %s ###\n", serial.GetCTS()?"on":"off");
}

// Handle DSR event
if (eEvent & CSerial::EEventDSR){
   printf("\n### Data set ready %s ###\n", serial.GetDSR()?"on":"off");
}

// Handle error event
if( eEvent & CSerial::EEventError ){
   printf( "\n### ERROR: " );
   switch( serial.GetError( ) ){
   case CSerial::EErrorBreak:
      printf("Break condition");
      break;
   case CSerial::EErrorFrame:
      printf("Framing error"),
      break;
   case CSerial::EErrorIOE:
      printf("IO device error");
      break;
   case CSerial::EErrorMode:
      printf("Unsupported mode");
      break;
   case CSerial::EErrorOverrun:
      printf("Buffer overrun");
      break;
   case CSerial::EErrorRxOver:
      printf("Input buffer overflow"),
      break;
   case CSerial::EErrorParity:
      printf("Input parity error");
      break;
   case CSerial::EErrorTxFull:
      printf("Output buffer full");
      break;
   default:
      printf("Unknown");
      break;
   }
   printf(" ###\n");
}

// Handle ring event
if (eEvent & CSerial::EEventRing){
   printf("\n### RING ###\n");
}

// Handle RLSD/CD event
if (eEvent & CSerial::EEventRLSD){
   printf("\n### RLSD/CD %s ###\n", serial.GetRLSD()?"on":"off");
}

// Handle data receive event
if (eEvent & CSerial::EEventRecv){
   // Read data, until there is nothing left
   do{
      // Read data from the COM-port
      lLastError   serial.Read(szBuffer, sizeof(szBuffer)-1, &dwBytesRead);

      // Verifying if there is error in transmition
```

```
            if (lLastError != ERROR_SUCCESS)
               return ::ShowError(serial.GetLastError(), _T("Unable to read from COM-
port."));

            if (dwBytesRead > 0){
               // Finalize the data, so it is a valid string
               szBuffer[dwBytesRead]   '\0';

               // Transfering the read data to a fixed buffer
               for(int i = 0; i < (int)dwBytesRead; i++){
                  if( ( szBuffer[i] != ',' ) && ( szBuffer[i] != '\r' ) ){
                       Data[j] - szBuffer[i];
                       j++;
                  }
                  else{
                       Sensor[sensnum] = atoi(Data);
                       sensnum++;
                       for(int y=0; y<50;y++){
                               Data[y] = '\0';
                       }
                       j = 0;
                  }
               }

               // Check if EOF (CTRL+'[') has been specified
               if( strchr( szBuffer, EOF_Char ) )
                  fContinue = false;
            }
         } while( dwBytesRead == sizeof( szBuffer) - 1 ); // End receiving
      } // End of receive event
   } while( fContinue ); // Finish waiting for any event
}

//---------------------------------------------------------------------------
// Closes the serial port
int SerialClose( void ){

   serial.Close();

   return 0;
}

//---------------------------------------------------------------------------
// Shows error message
int ShowError( LONG lError, LPCTSTR lptszMessage ){

   // Generate a message text
   TCHAR tszMessage[256];
   wsprintf(tszMessage,_T("%s\n(error code %d)"), lptszMessage, lError);

   // Display message-box and return with an error-code
   ::MessageBox(0,tszMessage,_T("Fatal Error"), MB_ICONSTOP|MB_OK);
   return 1;
}

////////////////////////////////////////////////////////////////////////////
// Fuzzy functions

//---------------------------------------------------------------------------
// Fuzzyfies the clustered values of inputs and returns the system state mu[i]
void Fuzzyfi( void ){
   float centref = (float)CENTREF;
   float centren = (float)CENTREN;
   float sigma  = (float)SIGMA;

   for( int j = 0; j < SENSORNUMBER; j++ ){
      X[j] = ( ( (float)Sensor[j] ) / ( (float)RANGE ) );
   }
```

```
// Clustering the sensors signals
XCluster[0] = (float)( ( X[3] + X[4] + X[5] + X[6] ) / 4 );      // Front sensing
XCluster[1] = (float)( ( X[1] + X[8] + X[9] ) / 3 );            // Back sensing
XCluster[2] = (float)( ( X[1] + X[2] + X[3] ) / 3 );            // Left sensing
XCluster[3] = (float)( ( X[6] + X[7] + X[8] ) / 3 );            // Right sensing

/* Fuzzyfing the inputs */
mu[0] = Mul(   Gauss_far( XCluster[0], centref, sigma ),
           Gauss_far( XCluster[1], centref, sigma ),
           Gauss_far( XCluster[2], centref, sigma ),
           Gauss_far( XCluster[3], centref, sigma ) );
mu[1] = Mul(   Gauss_far( XCluster[0], centref, sigma ),
           Gauss_far( XCluster[1], centref, sigma ),
           Gauss_far( XCluster[2], centref, sigma ),
           Gauss_near( XCluster[3], centren, sigma ) );
mu[2] = Mul(   Gauss_far( XCluster[0], centref, sigma ),
           Gauss_far( XCluster[1], centref, sigma ),
           Gauss_near( XCluster[2], centren, sigma ),
           Gauss_far( XCluster[3], centref, sigma ) );
mu[3]   Mul(   Gauss_far( XCluster[0], centref, sigma ),
           Gauss_far( XCluster[1], centref, sigma ),
           Gauss_near( XCluster[2], centren, sigma ),
           Gauss_near( XCluster[3], centren, sigma ) );
mu[4]   Mul(   Gauss_far( XCluster[0], centref, sigma ),
           Gauss_near( XCluster[1], centren, sigma ),
           Gauss_far( XCluster[2], centref, sigma ),
           Gauss_far( XCluster[3], centref, sigma ) );
mu[5] = Mul(   Gauss_far( XCluster[0], centref, sigma ),
           Gauss_near( XCluster[1], centren, sigma ),
           Gauss_far( XCluster[2], centref, sigma ),
           Gauss_near( XCluster[3], centren, sigma ) );
mu[6] = Mul(   Gauss_far( XCluster[0], centref, sigma ),
           Gauss_near( XCluster[1], centren, sigma ),
           Gauss_near( XCluster[2], centren, sigma ),
           Gauss_far( XCluster[3], centref, sigma ) );
mu[7] = Mul(   Gauss_far( XCluster[0], centref, sigma ),
           Gauss_near( XCluster[1], centren, sigma ),
           Gauss_near( XCluster[2], centren, sigma ),
           Gauss_near( XCluster[3], centren, sigma ) );
mu[8] = Mul(   Gauss_near( XCluster[0], centren, sigma ),
           Gauss_far( XCluster[1], centref, sigma ),
           Gauss_far( XCluster[2], centref, sigma ),
           Gauss_far( XCluster[3], centref, sigma ) ),
mu[9] = Mul(   Gauss_near( XCluster[0], centren, sigma ),
           Gauss_far( XCluster[1], centref, sigma ),
           Gauss_far( XCluster[2], centref, sigma ),
           Gauss_near( XCluster[3], centren, sigma ) );
mu[10] = Mul(  Gauss_near( XCluster[0], centren, sigma ),
           Gauss_far( XCluster[1], centref, sigma ),
           Gauss_near( XCluster[2], centren, sigma ),
           Gauss_far( XCluster[3], centref, sigma ) );
mu[11] = Mul(  Gauss_near( XCluster[0], centren, sigma ),
           Gauss_far( XCluster[1], centref, sigma ),
           Gauss_near( XCluster[2], centren, sigma ),
           Gauss_near( XCluster[3], centren, sigma ) );
mu[12] = Mul(  Gauss_near( XCluster[0], centren, sigma ),
           Gauss_near( XCluster[1], centren, sigma ),
           Gauss_far( XCluster[2], centref, sigma ),
           Gauss_far( XCluster[3], centref, sigma ) );
mu[13]   Mul(  Gauss_near( XCluster[0], centren, sigma ),
           Gauss_near( XCluster[1], centren, sigma ),
           Gauss_far( XCluster[2], centref, sigma ),
           Gauss_near( XCluster[3], centren, sigma ) );
mu[14] = Mul(  Gauss_near( XCluster[0], centren, sigma ),
           Gauss_near( XCluster[1], centren, sigma ),
           Gauss_near( XCluster[2], centren, sigma ),
           Gauss_far( XCluster[3], centref, sigma ) );
mu[15] = Mul(  Gauss_near( XCluster[0], centren, sigma ),
           Gauss_near( XCluster[1], centren, sigma ),
```

```
                Gauss_near( XCluster[2], centren, sigma ),
                Gauss_near( XCluster[3], centren, sigma ) );
}

//-------------------------------------------------------------------------
// Returns the minimum of the set
float Min( float op_A, float op_B, float op_C, float op_D ){
   if( op_A > op_B ) return( op_B );
   else return( op_A ),
}

//-------------------------------------------------------------------------
// Returns the product of the set
float Mul( float op_A, float op_B, float op_C, float op_D ){
   return (float)(op_A * op_B * op_C * op_D);
}

//-------------------------------------------------------------------------
// Returns the maximum of the setfloat max(float x, float y);
float Max( float op_A, float op_B ){
   if( op_A > op_B ) return( op_A );
   else return( op_B );
}

//-------------------------------------------------------------------------
// Gaussian membership function for far distances
float Gauss_far( float x, float c, float s ){
   float tmp1, tmp2;

   if( x <= c ) return 1.0;
   else{
      tmp1 = ( x   c ) / s;
      tmp2 = ( float )( ( -0.5 ) * tmp1 * tmp1 ),
      return ( float )exp( tmp2 );
   }
}

//-------------------------------------------------------------------------
// Gaussian membership function for near distances
float Gauss_near( float x, float c, float s ){
   float tmp1, tmp2;

   if( x >= c ) return 1.0;
   else{
      tmp1 = ( x   c ) / s;
      tmp2 = ( float )( ( -0.5 ) * tmp1 * tmp1 );
      return ( float )exp( tmp2 );
   }
}

////////////////////////////////////////////////////////////////////////////
// Reinforcement functions
//-------------------------------------------------------------------------
// Main reinforcement learning function
void Reinforce(void){

   // Resetting the external reinforcement signal
   r[0]   0;
   r[1]   0;
   r[2]   0;
   r[3] = 0;

   // Critic
   for( int i = 0; i < AGENTNUM; i++){
      p_old[i]   p[i];
      p[i] = 0;
      for( int j = 0; j < 16; j++ ){
         // Prediction of eventual reinforcement
```

```
            p[i] += mu[j] * v[i][j];
        }
    }

    // Finding the external reinforcement signal
    for( int i = 0; i < AGENTNUM; i++){
        if( XCluster[i] >= MAXPROX )    r[i] = -1;
    }

    for( int i = 0; i < AGENTNUM; i++){
        // Internal reinforcement signal
        r_hat[i] = r[i] + (float)( GAMMA * p[i] )   p_old[i];
    }

    // Calculating the square error
    for( int i = 0; i < AGENTNUM; i++ ){
        E[i] = (float)(0.005 * r_hat[i] * r_hat[i]),
    }

    // Calculating the probability of success for each agent
    for( int i = 0; i < AGENTNUM; i++ ){
        for( int j = 0; j < 16; j++ ){
            ro[i][j] = s_shape( w[i][j] );
        }
    }

    for( int i = 0; i < AGENTNUM; i++ ){
        temp[0] = 0;
        temp[1] = 0;
        for( int j   0; j < 16; j++ ){
            temp[0] += ro[i][j] * mu[j];
            temp[1] += mu[j];
        }
        P[i] = temp[0] / temp[1];
    }

    // Assigning the selected behaviours
    temp[0] = 0;
    temp[1] = 0;
    for(int i = 0; i < AGENTNUM; i++ ){
        temp[0] += (float)behaviour[i][0] * (P[i] );
        temp[1] += (float)behaviour[i][1] * (P[i] );
    }

    //// Calculating speed of the wheels
    LeftSpeed = (int)(SPEEDMAX * temp[0]);
    RightSpeed = (int)(SPEEDMAX * temp[1]);

    // Reinforce values... finding the weights
    for( int i = 0; i < AGENTNUM; i++){
        for( int j = 0; j < 16; j++ ){
            w[i][j]+= (float)ALPHA * r_hat[i] * mu[j] * (1/ro[i][j]) * (exp(-w[i][j]));
            v[i][j]-= (float)BETA * r_hat[i] * mu[j];

            if( w[i][j] >= 5.0 )          w[i][j] = 5.0;
            if( w[i][j] <= -5.0 )         w[i][j] = -5.0;
            if( v[i][j] >= 10.0 )         v[i][j] = 10.0;
            if( v[i][j] <= -10.0 )        v[i][j] = -10.0;
        }
    }
    PrintMatrix( 'P' );
}

//-------------------------------------------------------------------------
// S-function implementation
float s_shape( float s ){
    return (float)( 1 / ( 1.0 + (float)exp( -s ) ) );
}
```

```
//-------------------------------------------------------------------------
// Screen writing functions for certain values
void PrintMatrix( char select ){

   switch( select ){
   case 'w':
      printf("\nW");
      for( int i = 0; i < 4; i++ ){
         printf("\n\nThis is r_hat%i: %f", i, r_hat[i]);
         printf("\nAgent %i:\n", i);
         for( int j = 0; j < 16; j++ ){
            printf("   %f", w[i][j]);
         }
      }
      break;
   case 'v':
      printf("\nV");
      for( int i = 0; i < 4; i++ ){
         printf("\n\nThis is r_hat%i: %f", i, r_hat[i]);
         printf("\nAgent %i:\n", i);
         for( int j = 0; j < 16; j++ ){
            printf("   %f", v[i][j]);
         }
      }
      break;
   case 'r':
      printf("\nro");
      printf("\n av spd:%f", (((float)LeftSpeed + (float)RightSpeed) / 2));
      for( int i = 0; i < 4; i++ ){
         printf("\n\nThis is r_hat%i: %f", i, r_hat[i]);
         printf("\nAgent %i:\n", i);
         for( int j = 0; j < 16; j++ ){
            printf("   %f", ro[i][j]);
         }
      }
      break;
   case 'P':
      for( int i = 0; i < 4; i++ ){
         printf("\nP[%i]: %f", i, P[i]);
      }
      printf("\n");
      break;
   case 'S':
      printf("Left: %i Right: %i\n", LeftSpeed, RightSpeed);
      break;
   }
}
```